Treball de Fi de Màster

# Estudi i desenvolupament d'un *crypter*

Oriol Castejón

Màster Universitari en Seguretat de les
Tecnologies de la Informació i de les Comunicacions

Universitat Oberta de Catalunya

Juny de 2019

# Índex

# Capítol 1

# Introducció

Una de les coses més importants pels desenvolupadors de *malware* és que els programes antivirus i diferents eines de seguretat no puguin detectar els seus programes maliciosos. Per a aconseguir-ho, una de les opcions és l'ús de *crypters* o *packers*: programes que xifren un executable, afegint-hi les pròpies rutines necessàries per al desxifrat que només s'usaran en el moment d'execució. Així, el *malware* desencriptat només és present en la memòria RAM del sistema i per tant es pot evitar la seva detecció mitjançant signatures. L'objectiu final de tot *crypter* és la creació d'un arxiu *Fully Undetectable* o FUD, és a dir, que no sigui detectat per cap antivirus.

En aquest treball ens proposem el desenvolupament d'un *crypter*. L'interès que pot tenir construir-ne un, des d'un punt de vista de la protecció de sistemes informàtics, té múltiples vessants. Per una banda, ens permetrà entendre tècniques molt comunes usades per *malware* actual. Entendre aquestes tècniques amb profunditat és clau per saber els seus punts forts (i per tant, quins aspectes defensiu s'han de potenciar), així com per entendre els seus punts febles (és a dir, de quina manera es poden detectar i mitigar). A més, el desenvolupament d'aquesta eina també ens permetrà aprofundir el coneixement sobre el funcionament intern de Windows: caldrà que tinguem una idea ben clara de l'estructura que tenen els executables de Windows (el format *Portable Executable* o PE), i també haurem d'usar funcions de l'API de Windows, que ens permetrà entendre quina funcionalitat ofereix aquest sistema operatiu en una capa diferent a la del simple usuari o administrador de sistemes. També, el fet de crear un *crypter* propi ens permetrà entendre amb quina facilitat o dificultat es poden crear arxius maliciosos que no siguin detectats per antivirus, i com a conseqüència, ser conscients de quins són els límits que tenen aquestes eines. Finalment, tenir un *crypter* propi també ens pot servir per posar a prova els nostres sistemes davant d'un *malware* que no estigui estès, ser usat en tests de penetració, etc.

En aquest context, el programa que executa les rutines de desxifratge i executa l'arxiu maliciós s'anomena *stub*. Tenint en compte que aquest és un dels elements claus de qualsevol *crypter*, el *crypter* que hem creat ha estat anomenat `stubborn`. Aquest està pensat com una aplicació web empaquetada dins un contenidor de Docker. A grans

trets, la interfície d'usuari és un formulari web on es poden configurar certes opcions, pujar l'executable maliciós i obtenir un *stub* que conté una versió xifrada d'aquest arxiu. L'aplicació web està escrita en Python, mentre que l'*stub* està escrit en C++. Tot el codi és públic i està penjat en el repositori de GitHub [7].

L'estructura del treball és la següent. En el Capítol 2 es proporciona una visió general de l'estat de l'art dels *crypters*, tant en l'ús que se'n fa actualment, com exemples concrets de certa rellevància. A continuació, en el Capítol 3 descriurem el format d'executables de Windows, és a dir, el format *Portable Executable* o PE. Després, en el Capítol 4 farem una descripció a alt nivell de l'*stub* que crearem. Un cop fet això, passarem a descriure les tècniques usades amb més profunditat: per una banda, la tècnica coneguda com a *Process Hollowing* en el Capítol 5, així com diferents tècniques d'evasió d'antivirus en el Capítol 6. Tenint tots aquests elements ja podrem passar a descriure l'estructura del *crypter*, és a dir, l'aplicació web que hem construït, en el Capítol 7. Finalment, en el Capítol 8 presentarem els resultats de l'anàlisi d'un arxiu maliciós empaquetat amb el nostre *crypter*. Compararem, per una banda, els resultats obtinguts sense usar cap tècnica d'evasió d'antivirus amb els resultats obtinguts quan s'hagin usat aquestes tècniques descrites anteriorment. Per acabar, en el Capítol 9 presentarem les conclusions d'aquest treball, així com possibles vies de treball futur.

# Capítol 2

# Estat de l'art

En aquest capítol descriurem quin és l'ús actual dels *crypters* i revisarem alguns exemples concrets. L'objectiu principal no és donar una visió exhaustiva de tots els *crypters* i tècniques que hi ha, si no que és, per una banda, entendre la rellevància que tenen els *crypters* en el context del cibercrim actual i, per altra, posar certs exemples concrets de certa rellevància, que a més ens permetin fer-nos una idea de la varietat de *crypters* que es poden trobar actualment.

## 2.1 Ús de *crypters*

En l'àmbit del cibercrim i, en particular, en el *malware*, l'ús de *packers* o *crypters* és generalitzat: segons [3], [24], més del 80% dels virus informàtics usen alguna tècnica de *packing*. Segons dades de l'any 2006, més d'un 50% de les noves mostres de virus informàtics que s'obtenien no eren més que *malware* ja existent empaquetat amb algun *packer* diferent [32]. L'explicació d'aquest fet és la dificultat que tenen els antivirus per detectar *malware* que estigui empaquetat de forma genèrica, que implica que la manera més fàcil que hi ha de crear un *malware* nou és simplement empaquetar-ne un d'existent.

Efectivament, les empreses d'antivirus responen a l'aparició de nous *packers* o *crypters* detectant-ne la manera que aquests desempaqueten el codi maliciós i l'executen. No obstant, això fa que nous *packers* es vagin desenvolupant contínuament, que seran usats fins que cridin l'atenció dels antivirus i n'implementin un mètode de detecció. En aquest context, doncs, és interessant saber i entendre com de difícil és el procés de creació d'un *crypter* nou que no detecti cap antivirus.

També és interessant destacar que hi ha un negoci *underground* al voltant dels *packers*. El valor d'un *crypter* acaba venint determinat per les seves capacitats d'evasió d'antivirus: si l'*stub* creat és totalment indetectable (*Fully Undetectable* o FUD), ja sigui perquè encara no ha estat prou usat (i per tant no ha entrat en el punt de mira de les empreses d'antivirus) o bé perquè usa tècniques sofisticades (com el polimorfisme, en el que el codi

de l'*stub* mai és el mateix, però el resultat sempre és equivalent), estarà més cotitzat que d'altres. Per exemple, l'any 2014 es podia comprar en fòrums brasilers una llicència per un *crypter* totalment indetectable (FUD) per uns 19$ al mes, incrementant el preu a 29$-39$ si aquest tenia característiques addicionals, mentre que podia baixar a uns 10$ si el *crypter* no era FUD [25]. De manera similar, segons [16], en els fòrums *underground* russos, l'any 2012 un *crypter* costava uns 10$-15$ de mitjana, on el preu també experimentava bastanta variabilitat segons les característiques d'aquest: per exemple, un *crypter* polimòrfic podia costar més de 100$. Tot i aquests preus assequibles, la demanada fa que el negoci sigui profitós: per exemple, l'any 2011 s'oferia feina per a desenvolupadors de *crypters* en fòrums russos *underground*, amb sous que podien anar dels 2000$ als 5000$ mensuals [22].

Finalment cal destacar l'existència de diversos *packers* el codi dels quals es pot trobar a Internet que els diferents antivirus solen detectar. No obstant, aquest codi es pot modificar de forma relativament fàcil (afegint capes de xifrat, modificant els algorismes, etc.) de manera que els arxius xifrats puguin evadir la detecció d'antivirus. Segons [32], l'any 2006 es creaven uns 10 o 15 d'aquest tipus de *packers* modificats al mes.

En resum, doncs, podem dir que els *packers* o *crypters* estan contínuament en evolució, i a més són una peça fonamental de l'esfera del cibercrim, tant per l'ús que se'n fa com pel negoci que generen. Per tant, entendre el seu funcionament, la seva complexitat i les facilitats o dificultats que es troben per evitar la seva detecció és de vital importància per saber fins a quin punt les eines antivirus poden ser efectives.

## 2.2 Exemples de *crypters*

Com s'ha comentat en l'apartat anterior, el nombre de *crypters* existents és il·limitat: contínuament se'n van desenvolupant de nous i modificant d'antics per tal de no ser detectats per antivirus. En aquesta secció, però, posarem alguns *crypters* que hem trobat rellevants, explicant-ne algunes de les seves característiques.

Dels primers *crypters* que podem destacar és l'anomenat Yoda's Protector [37], ja que és dels més antics que es poden trobar a Internet actualment. Com podem veure a la pàgina web, es tracta d'un *crypter* gratuït i de codi obert per a executables de Windows de 32 bits. Suporta arxius de tipus .exe, .dll, .ocx i .scr, i, entre d'altres, implementa polimorfisme i tècniques d'*antidebugging*.

Un altre exemple de *crypter* de codi obert és Hyperion [1]. Es tracta d'una aplicació de línia de comandes desenvolupada en C/C++. El que és rellevant d'aquest *crypter* és el fet que evita tenir un executable "contenidor" precompilat al qual se li injecta el *malware* encriptat, ja que això el fa poc estable: s'han de modificar manualment capçaleres, adreces, etc., i considerar tots els casos possibles és difícil. En canvi, Hyperion usa el codi en ensamblador del contenidor, que es converteix en executable usant l'ensamblador `fasm` [15]. L'avantatge d'això és que la modificació de les capçaleres i adreces és duta a terme de forma automàtica per `fasm`, de manera que s'eviten tots els problemes que podrien

succeir en fer-ho manualment i es simplifica molt la tasca. Aquest *crypter* ha servit d'inspiració per crear el *crypter* d'aquest treball, tot i que en el nostre cas no usarem el llenguatge ensamblador directament, si no que treballarem amb C++.

També hi ha exemples comercials, per exemple l'anomenat core-packer de Hacking Team, una empresa italiana de desenvolupament de software d'espionatge. El codi d'aquest *crypter* es va fer públic amb les filtracions que van afectar aquesta companyia l'any 2015, i actualment es pot trobar a GitHub [10]. Aquest *crypter*, a diferència dels anteriors, admet executables de 32 i 64 bits. També usa diverses tècniques d'ofuscació, per exemple l'ús de la funció `GetProcAddress` de l'API de Windows amb cadenes text polimòrfic [11]. En poques paraules, aquesta tècnica permet amagar les funcions importades davant l'anàlisi estàtica de l'arxiu, i dificultar-ne la seva identificació durant una anàlisi dinàmica. En el nostre *crypter* usarem una tècnica similar (vegeu la Secció 6.1). D'aquest *crypter* també és destacable el fet que les implementacions dels algorismes criptogràfics d'aquest *packer* tenen certes flaqueses que permeten detectar-lo de forma heurística [35].

També hi ha *crypters* que han estat creats com a experiments per a testejar l'efectivitat d'antivirus, com el que ens proposem en aquest treball, com per exemple peCloak [12], escrit en Python. Un altre exemple és [33], treball de recerca en el qual s'investiguen diferents mètodes per a la creació de *crypters* i es compara la seva efectivitat alhora d'evadir la detecció d'antivirus.

Com a conclusió, doncs, podem dir que el món dels *crypters* és molt variat. Per una banda, per les motivacions que hi pot haver darrere: tot i que hi ha un negoci important, també hi ha implementacions de codi obert, així com recerca des d'un àmbit més acadèmic. D'altra banda, també hem vist que hi ha una gran varietat d'implementacions, variacions i llenguatges usats per a la seva creació.

# Capítol 3

# El format PE

Per tal d'entendre com funciona un *crypter* és necessari entendre prèviament quina és l'estructura bàsica d'un executable. Així doncs, en aquesta secció descriurem el format de Microsoft *Portable Executable* (PE), usat, entre d'altres, per als executables, DLLs i *drivers* de Windows. L'objectiu, però, no és fer-ne una descripció exhaustiva, si no centrar-nos en aquells aspectes que després resultaran ser rellevants per a la creació del *crypter* i, més concretament, en la tècnica de *Process Hollowing*. Per a una descripció exhaustiva d'aquest format referim al lector a la documentació de Microsoft MSDN [21], [30], o bé d'altres explicacions que es poden trobar, com per exemple [2], [4], [6], [31].

El format PE està basat és un extensió del format COFF (Common Object File Format), usat en sistemes Unix, i està construït de tal manera que sigui compatible amb l'antic format executable DOS MZ (el format que tenien els arxius .exe en els sistemes DOS). Com altres formats d'arxius executables, consisteix en una sèrie de dades estructurades en capçaleres o *headers* localitzades en posicions conegudes o fàcils de calcular, que descriuen el que conté la resta de l'arxiu, que està organitzat en seccions. D'aquesta manera el *loader* de Windows pot carregar l'arxiu en memòria i determinar quines parts s'han d'executar, com ha de resoldre les seves dependències, etc. Aquest format va ser adoptat per Microsoft a partir del sistema operatiu Windows NT 3.1.

A continuació passem a descriure les parts principals de l'estructura d'un arxiu PE, centrant-nos sempre en els arxius executables i deixant de banda les característiques dels arxius objecte.

## 3.1 L'*stub* DOS

El primer que trobem en un PE és, de fet, un programa per a ser executat en MS-DOS. Aquest programa o *stub* està inclòs en el format per tal de tenir compatibilitat cap enrere, per tal que qualsevol executable de Windows pugui ser executat en MS-DOS. Com veurem, però, aquest petit programa no té cap rellevància. A continuació descrivim les parts que

el formen.

**Capçalera DOS**
Com indica el nom, conté les metadades del programa DOS. En realitat, l'únic que ens importa és el valor guardat en la posició posició 0x3c, anomenat `e_lfanew`. Aquest valor indica l'adreça relativa de l'inici de la capçalera PE real i és, per tant, essencial per tal de poder accedir a la informació que permetrà carregar l'arxiu PE real en memòria.

**Stub**
A continuació hi trobem les dades del programa DOS, és a dir, allò que s'executaria si provéssim d'executar-lo en MS-DOS. En la pràctica totalitat dels casos, aquest *stub* només imprimeix el missatge `This program cannot be run in DOS mode`, que és l'*stub* per defecte que usa el *linker* de Windows.

## 3.2 Les capçaleres PE

A continuació trobem les capçaleres PE. Aquestes estan formades per les següents parts:

**Signatura**
El primer que es troba després de l'*stub*, a la posició indicada pel valor `e_lfanew` descrit anteriorment, és la signatura PE que identifica l'arxiu com un Portable Executable. Aquesta signatura és simplement és el valor 0x00004550, és a dir el valor ASCII en hexadecimal de les lletres "P" i "E" seguides de dos bytes nuls, en *little-endian*.

**Capçalera d'arxiu COFF**
A continuació, es troba un *COFF File Header* estàndard. En aquesta capçalera hi trobem informació com el tipus de CPU en el qual s'ha d'executar l'arxiu (`Machine`), el nombre de seccions de l'arxiu (`NumberOfSections`) o la mida de la capçalera opcional (`SizeOfOptionalHeader`), entre d'altres.

**Capçalera opcional**
Finalment, trobem la capçalera opcional. A diferència del que sembla indicar el nom, aquesta capçalera no és opcional en absolut en el cas d'executables PE, ja que conté la informació essencial per tal que el *loader* de Windows pugui carregar la imatge en memòria. Aquesta capçalera es pot dividir en tres parts:

- **Camps estàndard**: són aquells que estan en totes les implementacions del format COFF. Entre aquests camps hi trobem informació com si l'espai d'adreces permès és de 32 o 64 bits (en el camp `Magic`), la mida total de les seccions que contenen codi (`SizeOfCode`), l'adreça que s'ha d'executar quan l'arxiu es carregui en memòria

(`AddressOfEntryPoint`) o l'adreça relativa de la secció que conté el codi respecte la base de la imatge, un cop aquesta s'hagi carregat en memòria (`BaseOfCode`). Notem que els PE de 32 bits (PE32) contenen el camp `BaseOfData` mentre que en els de 64 bits (PE32+) aquest camp és inexistent.

- **Camps específics de Windows**: són aquells necessaris per tal de suportar característiques concretes de Windows. Aquí hi trobem, per exemple, l'adreça preferida en la que s'hauria de carregar l'executable en memòria (`ImageBase`), la mida total de la imatge, incloent totes les capçaleres, un cop aquesta s'hagi carregat en memòria (`SizeOfImage`), o bé el nombre de directoris de dades (`NumberOfRvaAndSizes`).

- **Directoris de dades**: es tracta de parells d'adreces i mides de certes taules especials que es carregaran en memòria i que el sistema pot usar en temps d'execució. Cada directori de dades és un camp de 8 bytes definit de la següent manera:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

  El camp `VirtualAddress` indica l'adreça virtual relativa o RVA de la taula (és a dir, l'adreça respecte la base de la imatge un cop carregada en memòria). D'altra banda, el camp `Size` indica la mida d'aquesta taula.

## 3.3 Taula de seccions

Immediatament després de la capçalera PE es troba la taula de seccions. Cada entrada en aquesta taula es pot considerar, a tots els efectes, com la capçalera de la secció corresponent. Cada entrada consta de 40 bytes, entre els quals s'emmagatzema informació com el nom de la secció (`Name`), la mida total de la secció quan està carregada en memòria (`VirtualSize`), l'adreça relativa de la secció respecte la base de la imatge quan està carregada en memòria (`VirtualAddress`), la mida en disc de la secció (`SizeOfRawData`) o un punter a la primera pàgina de la secció (`PointerToRawData`). En el camp `Characteristics` també s'indica, entre d'altres, si aquesta secció té permisos d'execució, d'escriptura o de lectura.

## 3.4 Seccions

Típicament, un executable de Windows té diverses seccions predefinides. A continuació passem a resumir breument les més rellevants.

**La secció `.text`**
Possiblement la més rellevant, la secció `.text` és la que conté el codi executable. Típicament l'adreça del punt d'entrada `AddressOfEntryPoint` d'un PE apuntarà a aquesta secció.

**Les seccions `.bss`, `.rdata` i `.data`**
Aquestes seccions són les que contenen dades que usarà el codi. Per una banda, la secció `.bss` conté dades no inicialitzades (variables declarades com a estàtiques en una funció, per exemple). La secció `.rdata` conté dades de lectura (*read-only*), com per exemple *strings* o constants definides en el codi. Finalment, la secció `.data` conté la resta de variables globals (ja que les que no són globals es guarden a l'*stack* del procés).

**La secció `.rsrc`**
La secció `.rsrc` conté els recursos o *resources* que usa l'executable, que poden ser icones, fonts, menús, etc., però també altres executables. Les dades estan d'organitzades en una estructura d'arbre binari. Aquests recursos es poden carregar en el codi usant la funció de l'API de Windows `LoadResource`.

**Les seccions `.edata` i `.idata`**
La secció `.edata` conté les exportacions de l'executable o DLL, mentre que la secció `.idata` conté les importacions.

**La secció `.reloc`**
Aquesta secció conté la taula de relocalitzacions o *relocations*, és a dir, una taula que indica adreces que hauran de ser modificades en el moment de la càrrega de l'executable si l'adreça base de la imatge en memòria no és l'adreça preferida. La taula de relocalitzacions està dividida en blocs, on cadascun conté les relocalitzacions per una pàgina de 4K. La taula conté una entrada descrivint el bloc, seguit per les relocalitzacions que corresponen aquest bloc. Cada bloc està descrit mitjançant estructura que conté la RVA de la pàgina corresponent i la mida del bloc (el nombre total de relocalitzacions del bloc):

```
typedef struct BASE_RELOCATION_BLOCK {
    DWORD PageAddress;
    DWORD BlockSize;
} BASE_RELOCATION_BLOCK, *PBASE_RELOCATION_BLOCK;
```

D'altra banda, les entrades de les relocalitzacions estan descrites per una `WORD`. Els 4 bits més significatius d'aquesta `WORD` indiquen el tipus de relocalització, mentre que els 12 restants indiquen l'adreça relativa respecte l'inici de la pàgina que correspon al bloc. Cada entrada es pot descriure amb la següent estructura:

```
typedef struct BASE_RELOCATION_ENTRY {
    USHORT Offset : 12;
    USHORT Type : 4;
} BASE_RELOCATION_ENTRY, *PBASE_RELOCATION_ENTRY;
```

# Capítol 4

# Disseny de l'*stub*

La peça clau en la creació d'un *crypter* és la implementació de l'*stub*. Com s'ha dit anteriorment, aquest s'ha d'encarregar de desxifrar el *payload* o programa maliciós en memòria i executar-lo sense guardar-lo en disc. Per a fer-ho, però, cal determinar els següents punts:

- Com estarà empaquetat el *payload* dins l'*stub*. Podria ser una nova secció, estar inclòs com un recurs, etc.
- Com s'executa el *payload* un cop s'ha desxifrat. Recordem que no es pot guardar en un nou arxiu i executar-lo, ja que precisament el que intentem evitar és que el *payload* desxifrat toqui disc. Una opció és injectar-lo en un altre procés i per a fer-ho hi ha diverses tècniques entre les quals podem escollir:*Process Hollowing*, *DLL injection*, *Process Doppelgänging*, etc.

Per al nostre *stub* ens basarem en una de les idees exposades a [33] en el mètode denominat *Resource Packer*. Bàsicament, consisteix en empaquetar el *payload* xifrat com un recurs d'un executable, i en el moment d'execució carregar-lo en memòria mitjançant la funció de l'API de Windows `LoadResource`. Un cop en memòria, es pot desxifrar i injectar-lo en un altre procés. A diferència de [33], però, nosaltres no sobreescriurem el codi de l'*stub* en un executable "plantilla", com ells denominen, si no que crearem l'*stub* com un executable en sí mateix.

Un dels avantatges de seguir aquest mètode és que l'*stub* té l'aparença d'un executable normal i corrent (amb les seccions típiques dels arxius PE que hem descrit anteriorment), fet que el fa menys sospitós per a un antivirus. D'altra banda, el *payload* xifrat sol tenir una entropia elevada pel fet d'estar xifrat. No obstant, aquest no es guarda dins de seccions que normalment tenen poca entropia com la secció de `.text`, `.data`, etc., sinó a la secció `.rsrc` que, per la seva pròpia naturalesa, no té valors d'entropia estàndards. Per tant, aquest fet no pot ser utilitzat pels antivirus per a la detecció del nostre *stub* com a programa sospitós.

D'altra banda, el fet que haguem d'injectar el codi desxifrat en un procés extern comporta certa complexitat, ja que s'ha d'executar manualment certs passos que normalment fa el *loader* de Windows. La tècnica d'injecció de processos que hem escollit és la denominada *Process Hollowing*, que descriurem en la següent secció.

# Capítol 5

# La tècnica de *Process Hollowing*

La tècnica coneguda com a *Process Hollowing*, també anomenada a vegades RunPe o *Dynamic Forking*, consisteix a grans trets en crear un procés en estat suspès, modificar-ne el contingut per la imatge d'un altre executable, fer les modificacions necessàries per tal que l'execució de la nova imatge pugui dur-se a terme i finalment reiniciar l'execució del procés. D'aquesta manera, doncs, es pot executar codi maliciós en nom d'un procés conegut per l'usuari o el sistema. Aquest mètode, però, també permet executar codi xifrat sense haver de guardar una còpia desxifrada en disc: només cal fer-ho en memòria i injectar-lo en un altre procés qualsevol. És aquest fet, doncs, el que el fa adient per a la creació d'un *crypter*.

A continuació descriurem els diferents passos que cal seguir per tal d'executar aquesta tècnica, indicant en cada cas les funcions que proporciona l'API de Windows (incloent aquelles que no estan documentades) que es poden usar per dur-los a terme.

Tot i que aquesta tècnica ha estat implementada múltiples vegades (veure, per exemple, [23] o [17], per aquest treball hem fet la nostra implementació particular. Tot i que no inclourem el codi en aquesta memòria, sí que farem referència a les funcions que hem implementat i que es poden trobar al repositori de GitHub [7]. Els arxius rellevants per a aquesta secció són:

- `processHollower.cpp` i `processHollower.h`, que contenen la classe que implementa la tècnica del *Process Hollowing*, anomenada `ProcessHollower`.
- `windowsInternals.cpp` i `windowsInternals.h`, que contenen funcions auxiliars per a la manipulació d'estructures internes de Windows.

## 5.1 Càrrega del PE amfitrió

Abans de començar, cal carregar en memòria l'executable que volem injectar (l'amfitrió). Aquest pas no forma part del *Process Hollowing* pròpiament, però ens serveix per a obtenir

la informació necessària per a dur-lo a terme. El primer que farem serà guardar l'adreça on l'executable està carregat (`dwGuestPEAddress`), i així com carregar les seves capçaleres en un *struct* que anomenem `pGuestPEHeaders`. Aquesta estructura ens permetrà accedir posteriorment a diferents camps de les capçaleres del PE de forma fàcil. Aquest pas està implementat en el mètode `GetGuestPEData`.

## 5.2 Creació del procés amfitrió

Un cop fet això ja podem passar a crear el procés en mode suspès. Per això es pot usar la funció `CreateProcess` de l'API de Windows, passant la *flag* `CREATE_SUSPENDED`. La implementació d'aquest pas està feta en el mètode `CreateSuspendedProcess`. Aquí és important destacar que obtindrem l'atribut `lpProcessInformation` de la classe `ProcessHollower`, que conté la informació necessària del procés creat (per exemple un *handle* al procés, l'identificador del procés o PID, etc.).

## 5.3 Obtenció de l'adreça base del procés creat

A continuació passem a guardar certa informació del procés creat, concretament l'anomenat *Process Environment Block* (PEB). El PEB és una estructura opaca de Windows, poc documentada [29] i que està pensada per ser usada només pel sistema operatiu, però que ens permet accedir a dades del procés creat més fàcilment. Concretament, només ens interessa l'adreça base del procés creat, que guardarem en l'atribut `dwHostImageBaseAddress`. El mètode que s'encarrega de carregar el PEB i guardar l'adreça `dwHostImageBaseAddress` és `GetProcessData`, que crida a la funció auxiliar `ReadRemotePEB` (que es pot trobar a l'arxiu `windowsInternals.cpp`).

## 5.4 Desmapejat de la imatge del procés amfitrió

Un cop tenim l'adreça base del procés creat, no necessitem res més d'aquest procés i ja podem passar a desmapejar la memòria que s'havia reservat anteriorment. Això ho fem mitjançant el mètode `UnmapProcessMemory`, que crida la funció de l'API de Windows `NtUnmapViewOfSection`. Notem, però, que com que es tracta d'una funció no documentada, hem de fer la càrrega manualment. Per exemple, això es pot fer obtenint un *handle* a NTDLL (mitjançant `GetModuleHandleA`) i posteriorment usant la funció `GetProcAddress` d'aquest mòdul per trobar l'adreça de `NtUnmapViewOfSection`.

## 5.5 Reserva de memòria suficient per al PE amfitrió

A continuació passem a reservar la memòria necessària per poder carregar el PE amfitrió en el mètode `AllocateProcessMemory`. Com s'ha dit anteriorment, podem obtenir la mida que ocupa el PE amfitrió en memòria a partir del camp `SizeOfImage` de la capçalera opcional. A partir d'aquí, només cal que usem la funció de l'API de Windows `VirtualAllocEx`. En aquest moment, reservarem tota la memòria amb permisos de lectura, escriptura i execució. Posteriorment, un cop haguem escrit el contingut de cada secció, canviarem els seus permisos segons correspongui a cadascuna d'elles.

## 5.6 Còpia de la imatge del procés amfitrió

A continuació, doncs, ja podem passar a copiar les dades del PE dins de la memòria reservada en el procés amfitrió. Tot aquest procediment es fa en el mètode `InjectGuestPE`, que crida diversos cops al mètode `WriteProcessSection`.

En primer lloc copiem les capçaleres del PE amfitrió en el procés amfitrió usant la funció de l'API de Windows `WriteProcessMemory`. Notem que sabem l'adreça de destinació (donada per `dwHostImageBaseAddress`), l'adreça d'origen (donada per `dwGuestPEAddress`), així com la mida de les dades que hem de copiar (donades pel camp `SizeOfHeaders` de la capçalera opcional).

A continuació, es passa a copiar les diverses seccions (el nombre total ve donat pel camp `NumberOfSections` de la capçalera del PE amfitrió). Per això cal que tinguem en compte els camps de la taula de seccions (descrita anteriorment) següents: `VirtualAddress` (per saber l'adreça de destinació on s'ha de copiar la secció), `PointerToRawData` (per saber l'adreça d'origen des d'on copiar la secció) i `SizeOfRawData` (per saber la mida de la secció).

## 5.7 Aplicació de relocalitzacions

Abans de finalitzar la injecció del procés com a tal, caldrà aplicar les relocalitzacions si és necessari. Per això cal carregar el directori de dades de relocalitzacions, i anar aplicant cada bloc de relocalitzacions. Això es du a terme en el mètode `ApplyRelocations`, que és cridat dins del mètode `InjectGuestPE` just després d'escriure cada secció.

## 5.8 Canvi de permisos de les seccions

Un cop ja hem escrit tot el contingut de cada secció, podem donar els permisos que els correspon per tal de fer l'executable menys sospitós (ja que totes les seccions tinguin permisos de lectura, escriptura i execució no és gens usual). Per a fer-ho ens fixarem

en el camp `Characteristics` de la capçalera de cada secció, que ens permet determinar quins permisos han de tenir. El codi corresponent es pot trobar en el mètode `SetSectionsPermissions`. La funció que, a partir del camp `Characteristics`, determina els permisos de la secció és `GetMemProtectionFlag`, que es pot trobar en l'arxiu `winInternals.cpp`.

## 5.9 Canvi del punt d'entrada i represa de l'execució

Finalment, l'últim pas consisteix en establir el nou punt d'entrada del procés i continuarne l'execució, que es du a terme en el mètode `JumpToEntryPoint`. Per modificar el punt d'entrada, només cal carregar el context del procés (mitjançant la funció de Windows `GetThreadContext`) i modificar-ne el registre `EAX` (per executables de 32 bits) o el registre `RCX` (per a 64 bits). El valor que hi posarem l'obtenim a partir el camp `AddressOfEntryPoint` de la capçalera opcional del PE amfitrió, tenint en compte que és una adreça relativa i que per tant li hem de sumar `dwHostImageBaseAddress`. Un cop fet això, doncs, canviem el context del procés amb `SetThreadContext` i prosseguim l'execució del procés mitjançant la funció de l'API de Windows `ResumeThread`. Notem que no cal que resolem les importacions de l'executable amfitrió, ja que en reprendre l'execució el *loader* de Windows s'encarregarà de fer-ho.

# Capítol 6

# Tècniques d'evasió d'antivirus

Per tal d'evadir els antivirus que puguin detectar la nostra implementació de la tècnica de *Process Hollowing*, també usarem altres tècniques dedicades a emmascarar certes propietats de l'executable. Per una banda, usarem dues tècniques que consisteixen en donar a l'arxiu executable un aspecte menys sospitós: "amagant" les importacions de funcions sospitoses (les funcions de l'API de Windows típiques del *Process Hollowing*) [36], i per altra fent crides a l'API de Windows a funcions que no tenen cap efecte i que no encaixen dins d'aquesta tècnica (*junk API calls*) [33]. Aquestes tècniques són útils per evadir les deteccions basades en l'anàlisi estàtica de l'executable. Ara bé, si l'arxiu s'executa (o es simula la seva execució) per tal de determinar si és maliciós o no, necessitarem altres tipus de tècniques. En aquest cas, doncs, farem servir tècniques de detecció d'emulació de codi i detecció de *sandbox* (és a dir, un entorn aïllat especialment dissenyat per a l'anàlisi d'arxius). Aquestes tècniques consisteixen en fer certes comprovacions abans d'executar el codi maliciós per tal de determinar si l'arxiu està sent executat en un entorn fictici [19], [28], [33].

Cal dir que hi ha moltes variacions i implementacions d'aquestes tècniques, però aquí ens hem centrat en un nombre reduït. Seria interessant fer una comprovació de quines són més eficaces, si afegir moltes comprovacions pot aixecar sospites i tenir un efecte contrari al desitjat, etc.

Aquestes tècniques es poden trobar implementades en els següents arxius:

- `hiddenImports.cpp` i `hiddenImports.h`: Ofuscació d'importacions
- `junkApiCalls.cpp` i `junkApiCalls.h`: Crides falses a l'API de Windows
- `antivirusChecks.cpp` i `antivirusChecks.cpp`: Detecció d'emulació de codi i *sandbox*

## 6.1 Ofuscació d'importacions

A continuació explicarem com podem amagar les importacions de les funcions més sospitoses. Bàsicament, enlloc de cridar una funció directament, el mètode consisteix en resoldre l'adreça de cada importació en el moment d'execució. Per això podem usar les funcions `GetModuleHandle` o `LoadLibrary` per obtenir un *handle* a la DLL corresponent. Un cop es té aquest *handle*, es pot recórrer la taula d'exportacions per tal d'obtenir l'adreça de la funció desitjada. A més hem encriptat els noms de les DLLs i importacions, per tal que el no estiguessin en text clar en el codi de l'executable (ja que, si es així, es poden obtenir de forma molt fàcil mitjançant una anàlisi estàtica, per exemple amb el programa `strings`). Aquesta tècnica és bastant comuna en el *malware*, tot i que sovint s'usen *hashes* enlloc del xifratge (veure per exemple [10] o [20]).

Per a dur a terme aquest mètode primer hem creat una funció que, carrega una DLL i recorre la taula d'exportacions per tal de determinar l'adreça de la funció desitjada:

```
PVOID WINAPI GetFunctionAddress(LPCTSTR lpDllName,
 LPCSTR lpFunctionName) {
   PPE_HEADERS ppehModuleHeaders;
   HMODULE hModule;
   PDWORD pdwAddress, pdwName;
   PWORD pwOrdinal;

   hModule = GetModuleHandle(lpDllName);

   if (!hModule)
       hModule = LoadLibrary(lpDllName);
   if(!hModule)
       return nullptr;

   ppehModuleHeaders = LoadPEHeaders((VIRTUAL_ADDRESS)hModule);
   if(ppehModuleHeaders->NTHeaders->OptionalHeader.
     DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress == 0){
       return nullptr;
   }
   auto pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)hModule +
   ppehModuleHeaders->NTHeaders->OptionalHeader.
     DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
   pdwAddress = (PDWORD)((LPBYTE)hModule
     + pExportDirectory->AddressOfFunctions);
   pdwName = (PDWORD)((LPBYTE)hModule
     + pExportDirectory->AddressOfNames);
```

```
    pwOrdinal = (PWORD)((LPBYTE)hModule
      + pExportDirectory->AddressOfNameOrdinals);

    for(int i=0; i < pExportDirectory->AddressOfFunctions; i++) {
        if(!strcmp(lpFunctionName, (char*)hModule + pdwName[i])) {
            return (PVOID)((LPBYTE)hModule + pdwAddress[pwOrdinal[i]]);
        }
    }
    return nullptr;
}
```

Un cop fet això, només cal cridar aquesta funció amb les importacions desitjades. Com hem dit anteriorment, per a ser més cautelosos i no incloure els noms de les importacions a l'arxiu (que poden ser llegits de manera immediata amb qualsevol eina d'anàlisi estàtica), hem encriptat els noms de les funcions mitjançant l'algorisme XOR. Abans de cridar la funció anterior, doncs, caldrà desxifrar el nom corresponent. Les dues funcions següents contenen el codi per tal de cridar la funció `CreateProcessA` (la resta d'importacions no es mostren aquí):

```
PVOID WINAPI GetKernel32Function(LPCSTR lpFunctionName) {
    char encryptedKernel32[] =
    "\x0a\x07\x11\x0a\x00\x0a\x54\x5a\x47\x0e\x07\x0d";
    Xor(encryptedKernel32, 12, IMPORTS_KEY, IMPORTS_KEY_LENGTH);
    return GetFunctionAddress((LPCSTR)encryptedKernel32, lpFunctionName);
}

_CreateProcessA GetHiddenCreateProcessA() {
    char encryptedCreateProcessA[] =
    "\x22\x10\x06\x05\x11\x03\x37\x1a\x06\x09\x0e\x12\x11\x22";
    Xor(encryptedCreateProcessA, 14, IMPORTS_KEY, IMPORTS_KEY_LENGTH);
    auto HiddenCreateProcessA =
    (_CreateProcessA)GetKernel32Function((LPCSTR)encryptedCreateProcessA);
    return HiddenCreateProcessA;
}
```

En aquest cas, les dues variables `encryptedKernel32` i `encryptedCreateProcessA` contenen respectivament els texts `kernel32.dll` i `CreateProcessA` xifrats amb una clau definida en el propi arxiu.

En la Figura 6.1 es mostren, mitjançant l'eina d'anàlisi estàtica CFF Explorer, funcions importades abans i després d'aplicar aquesta tècnica. Si no apliquem la tècnica podem

veure fàcilment que s'usa la funció `CreateProcessA`, mentre que aplicant la tècnica d'ofuscació d'importacions aquesta funció desapareix de les importacions.

| OFTs | FTs (IAT) | Hint | Name |
|---|---|---|---|
| | | | |
| Qword | Qword | Word | szAnsi |
| 000000000001959C | 000000000001959C | 0056 | CloseHandle |
| 00000000000195AA | 00000000000195AA | 00AB | CreateProcessA |
| 00000000000195BC | 00000000000195BC | 00B5 | CreateSemaphoreW |
| 00000000000195D0 | 00000000000195D0 | 00D9 | DeleteCriticalSection |
| 00000000000195E8 | 00000000000195E8 | 00FA | EnterCriticalSection |
| 0000000000019600 | 0000000000019600 | 0159 | FindResourceA |

(a) Funcions importades d'un executable que usa `CreateProcessA`.

| OFTs | FTs (IAT) | Hint | Name |
|---|---|---|---|
| | | | |
| Qword | Qword | Word | szAnsi |
| 000000000001A644 | 000000000001A644 | 0056 | CloseHandle |
| 000000000001A652 | 000000000001A652 | 00B5 | CreateSemaphoreW |
| 000000000001A666 | 000000000001A666 | 00D9 | DeleteCriticalSection |
| 000000000001A67E | 000000000001A67E | 00FA | EnterCriticalSection |
| 000000000001A696 | 000000000001A696 | 0159 | FindResourceA |
| 000000000001A6A6 | 000000000001A6A6 | 0170 | FreeLibrary |

(b) Funcions importades del mateix executable que usa `CreateProcessA`, però usant la tècnica d'ofuscació d'importacions.

Figura 6.1: Comparació de les funcions importades abans i després d'aplicar la tècnica d'ofuscació d'importacions.

## 6.2 Crides falses a l'API de Windows

Com s'ha mencionat anteriorment, una de les altres tècniques que hem implementat ha estat fer crides innòcues a diferents funcions de l'API de Windows. Introduint aquestes crides entre les crides pròpies del *Process Hollowing* podem trencar regles heurístiques que considerin la execució de certes funcions en un cert ordre. Per altra banda, però, també ens permet incloure més importacions i DLLs usades a l'arxiu final, fet que ja de per sí farà que l'arxiu sigui menys sospitós.

Un exemple seria la crida de les funcions `GetActiveWindow` i `GetMenu`, que trobem a `USER.dll`:

```
VOID callGetMenu() {
```

```
    auto hWnd = GetActiveWindow();
    GetMenu(hWnd);
}
```

Un altre exemple seria la crida a la funció `IsTextUnicode`, continguda a `Advapi32.dll`:

```
VOID callIsTextUnicode() {
    const VOID *lpv = "is this text unicode?";
    int iSize = 21;
    auto lpiResult = new INT();
    IsTextUnicode(lpv, iSize, lpiResult);
}
```

Com es pot veure, aquestes funcions no tenen cap efecte, i per tant les podem anar cridant durant l'execució del *Process Hollowing*, sense que això variï el seu resultat.

En la Figura 6.2 podem veure les DLLs importades abans i després d'aplicar aquesta tècnica. Com es pot apreciar, en el primer cas només tenim dues DLLs (fet que fa més sospitós l'arxiu), mentre que en el segon cas en tenim 4 (amb una funció d'ADVAPI32.dll i 4 de USER.dll).

| Module Name | Imports | OFTs | TimeDateStamp | ForwarderChain | Name RVA | FTs (IAT) |
|---|---|---|---|---|---|---|
| | | | | | | |
| szAnsi | (nFunctions) | Dword | Dword | Dword | Dword | Dword |
| KERNEL32.dll | 49 | 0001903C | 00000000 | 00000000 | 00019BA0 | 000192EC |
| msvcrt.dll | 35 | 000191CC | 00000000 | 00000000 | 00019C3C | 0001947C |

(a) DLLs importades d'un executable que realitza *Process Hollowing*.

| Module Name | Imports | OFTs | TimeDateStamp | ForwarderChain | Name RVA | FTs (IAT) |
|---|---|---|---|---|---|---|
| | | | | | | |
| szAnsi | (nFunctions) | Dword | Dword | Dword | Dword | Dword |
| ADVAPI32.dll | 1 | 0001A064 | 00000000 | 00000000 | 0001ABB0 | 0001A34C |
| KERNEL32.dll | 49 | 0001A074 | 00000000 | 00000000 | 0001AC84 | 0001A35C |
| msvcrt.dll | 35 | 0001A204 | 00000000 | 00000000 | 0001AD20 | 0001A4EC |
| USER32.dll | 4 | 0001A324 | 00000000 | 00000000 | 0001AD3C | 0001A60C |

(b) DLLs importades d'un executable que realitza *Process Hollowing* afegint crides falses a l'API de Windows.

Figura 6.2: Comparació de les DLLS importades abans i després d'aplicar la tècnica de crides falses a l'API de Windows.

## 6.3 Detecció d'emulació de codi

Sovint, en entorns emulats hi ha certes funcions que es substitueixen per funcions que no fan res, sempre retornen el mateix, etc. Així doncs, una manera fàcil de comprovar si estem

en un d'aquests entorns és cridar aquestes funcions i determinar si el seu comportament no és l'esperat en un entorn normal.

Un exemple és la funció `Sleep`. En aquest cas, el que fan molts emuladors és substituir-la per una funció que no fa res, de manera que poden accelerar l'execució del codi de manera artificial. Per tant, podem cridar aquesta funció per tal que esperi un segon i comprovar el temps que ha passat entre just abans de cridar-la i just després. Si el temps no arriba a un segon, voldrà dir que aquesta funció no fa res i per tant ha estat emulada. La implementació que hem fet és la següent:

```
BOOL checkSleep() {
    DWORD dwCount1 = GetTickCount();
    Sleep(1000);
    DWORD dwCount2 = GetTickCount();
    if(dwCount2 - dwCount1 < 1000) {
        return TRUE;
    }
    return FALSE;
}
```

Un altre exemple és la funció `LoadLibraryA`, que permet carregar DLLs en la memòria del procés. Quan aquesta funció està emulada, pot ser que no retorni adreces per a DLLs reals (perquè no estan implementades), el qual seria un indicador que el codi està sent emulat. També, però, pot ser que sempre retorni una adreça, incloent quan es carrega una DLL falsa, el qual també ens estaria indicant l'emulació de la funció. Això ho podem comprovar amb el següent codi:

```
BOOL checkLoadDLL() {
    char const *realDLL[] = {"Kernel32.DLL", "networkexplorer.DLL",
                             "NlsData0000.DLL"};
    char const *falseDLL[] = {"NetProjW.DLL", "Ghofr.DLL", "fg122.DLL"};
    HMODULE hInstLib;
    for (int i = 0; i < 3; i++) {
        hInstLib = LoadLibraryA(realDLL[i]);
        if(hInstLib == nullptr) {
            return TRUE;
        }
        FreeLibrary(hInstLib);
    }
    for(int i = 0; i < 3; i++) {
        hInstLib = LoadLibraryA(falseDLL[i]);
        if(hInstLib != nullptr) {
            return TRUE;
```

```
        }
    }
    return FALSE;
}
```

## 6.4 Detecció de *sandbox*

Com s'ha explicat anteriorment, una *sandbox*, és un entorn aïllat específicament creat per a l'execució i anàlisi de *malware*. Sovint són màquines virtuals, amb característiques molt diferents a un sistema normal. Per tal de detectar si el codi està essent executat en una d'aquestes *sandboxs*, doncs, només cal fer certes comprovacions sobre les característiques del sistema (mitjançant la mateixa API de Windows) per determinar si estan fora del que és habitual.

Per exemple, podem comprovar si el sistema té menys de dos processadors. En cas que fos així indicaria que estem en un entorn no realista i per tant probablement seria una *sandbox*:

```
BOOL checkNumberOfProcessors() {
    SYSTEM_INFO siSysInfo;
    GetSystemInfo(&siSysInfo);
    if (siSysInfo.dwNumberOfProcessors < 2) {
        return TRUE;
    }
    return FALSE;
}
```

Un altre exemple seria comprovar la quantitat total de memòria. Si té menys d'1GB, consideraríem que estem en una *sandbox*:

```
BOOL checkPhysicalMemory() {
    MEMORYSTATUSEX mseMemoryStatus;
    mseMemoryStatus.dwLength = sizeof (mseMemoryStatus);
    GlobalMemoryStatusEx(&mseMemoryStatus);
    if ((mseMemoryStatus.ullTotalPhys/1024) < 1048576) {
        return TRUE;
    }
    return FALSE;
}
```

Finalment, podem comprovar la quantitat de disc lliure que hi ha. Si veiem que hi ha massa poc espai lliure, també podem considerar que es tracta d'un entorn sospitós:

```
BOOL checkDriveSize() {
    ULARGE_INTEGER total_bytes;
    if (!GetDiskFreeSpaceExA("C:\\", nullptr, &total_bytes, nullptr)) {
        return FALSE;
    }
    if (total_bytes.QuadPart / 1073741824 <= 60) {
        return TRUE;
    }
    return FALSE;
}
```

# Capítol 7

# Disseny del *crypter*

En aquesta secció explicarem com s'ha dissenyat i implementat el *crypter*, que hem anomenat `stubborn`. Tot el codi es pot trobar al repositori de GitHub [7]. Les característiques principals a nivell d'arquitectura i components són les següents:

1. El *crypter* està dissenyat com una aplicació web que corre dins d'un contenidor de Docker [13].

2. L'aplicació web està implementada amb el *microframework* de Python Flask [14].

3. El codi per l'*stub* està implementat en C++. Aquest codi es genera dinàmicament mitjançant `cookiecutter` [9], un mòdul de Python per a la generació de projectes a partir de plantilles.

4. Després de generar el codi per l'*stub*, aquest es compila mitjançant la versió del compilador `g++` de MinGW [27]. Per automatitzar tots els passos necessaris per a la compilació s'usa l'eina CMake [8].

## 7.1 Estructura de l'aplicació

El codi de l'aplicació està estructurat de la següent manera:

```
/
├── app/
│   ├── stub_templates/
│   │   └── process_hollowing/
│   ├── templates/
│   │   ├── _formhelpers.html
│   │   └── home.html
│   ├── tmp/
│   ├── app.py
│   ├── builder.py
│   ├── encryption.py
│   ├── form.py
│   └── requirements.txt
├── config/
│   ├── start.sh
│   └── stubborn-crontab
├── docker-compose.yml
└── Dockerfile
```

L'aplicació en sí està continguda en la carpeta `app`. La resta són arxius per a aixecar i configurar el contenidor de Docker. Concretament, fora d'aquesta carpeta només hi trobem:

- La carpeta `config`: Aquesta carpeta conté els arxius necessaris per a la configuració del contenidor de Docker. Concretament, l'*script* `start.sh` és el que bàsicament s'encarrega d'aixecar l'aplicació un cop s'arrenca el contenidor de Docker, mentre que l'arxiu `stubborn-crontab` serveix per definir les tasques de `cron` que correran dins el contenidor.

- Els arxius `docker-compose.yml` i `Dockerfile`: Mitjançant aquests arxius es defineix les propietats del contenidor de Docker.

Dins la carpeta `app` hi trobem el següent contingut:

- La carpeta `stub_templates`: Aquí és on estan contingudes les plantilles per a la creació de diferents *stubs*. Actualment, només hi ha la plantilla corresponent a la tècnica del *Process Hollowing*, dins la carpeta `process_hollowing`. Més endavant explicarem amb més detall el contingut d'aquesta carpeta.

- La carpeta `templates`: Aquí hi ha les plantilles HTML per al *frontend* de l'aplicació. El motor de plantilles que usa Flask és Jinja2 [18].

- La carpeta `tmp`: Inicialment buida, aquesta carpeta és on es guardarà el codi C++ generat dinàmicament i on es compilarà l'*stub*. Aquesta carpeta es va netejant periòdicament mitjançant una tasca de `cron`.

- Arxius Python: Són els arxius que implementen la lògica del *backend* de l'aplicació. Més endavant donarem més detalls de cada arxiu.
- Arxiu `requirements.txt`: És l'arxiu que conté les dependències de Python de l'aplicació.

## 7.2 Ús de l'aplicació

Per iniciar l'aplicació web només cal executar la següent comanda:

```
docker-compose up -d
```

Un cop fet això, ja tindrem el servei web escoltant al port 5000. Per accedir-hi només cal accedir a la URL http://127.0.0.1:5000 mitjançant un navegador, i obtindrem el formulari mostrat en la Figura 7.1. A continuació passem a descriure els camps:

- **Select Executable**: Permet seleccionar l'arxiu executable que es vol empaquetar dins de l'*stub*.
- **Target Executable**: Permet seleccionar l'executable per al qual es crearà el procés en estàs suspès, per tal de realitzar la tècnica de *Process Hollowing* i injectar-hi l'executable seleccionat en el camp anterior. Hi ha 4 opcions disponibles: el mateix *stub* (l'opció "Same stub file"), o bé els executables `calc.exe`, `notepad.exe` o `svchost.exe`. Cal notar que, en cas que l'executable original sigui de 64 bits, aquests executables s'obtindran de la carpeta `C:\Windows\System32`, mentre que si és de 32 bits s'obtindran de la carpeta `C:\Windows\SysWOW64`.
- **Build type**: Permet escollir quin el tipus de *build* de la compilació de l'*stub*. En el cas que s'esculli l'opció de *debug*, un cop s'executi l'*stub* es mostrarà en la consola informació sobre tots els passos que es van realitzant (Figura 7.2).
- **Key Options**: Permet escollir com es genera la clau amb què es xifrarà l'executable empaquetat: una clau aleatòria o bé una clau que esculli el propi usuari. En el primer cas es permet escollir la llargada de la clau, mentre que si s'escull la segona opció es mostrarà un quadre de text on l'usuari pot introduir la clau que s'ha d'usar. En qualsevol cas, s'usarà l'algorisme de xifratge XOR.

## 7.3 Backend

Quan un usuari omple el formulari anterior, s'envia una petició `POST` al *backend* de l'aplicació, que la gestiona i retorna la resposta adequada. Tot aquesta lògica està implementada

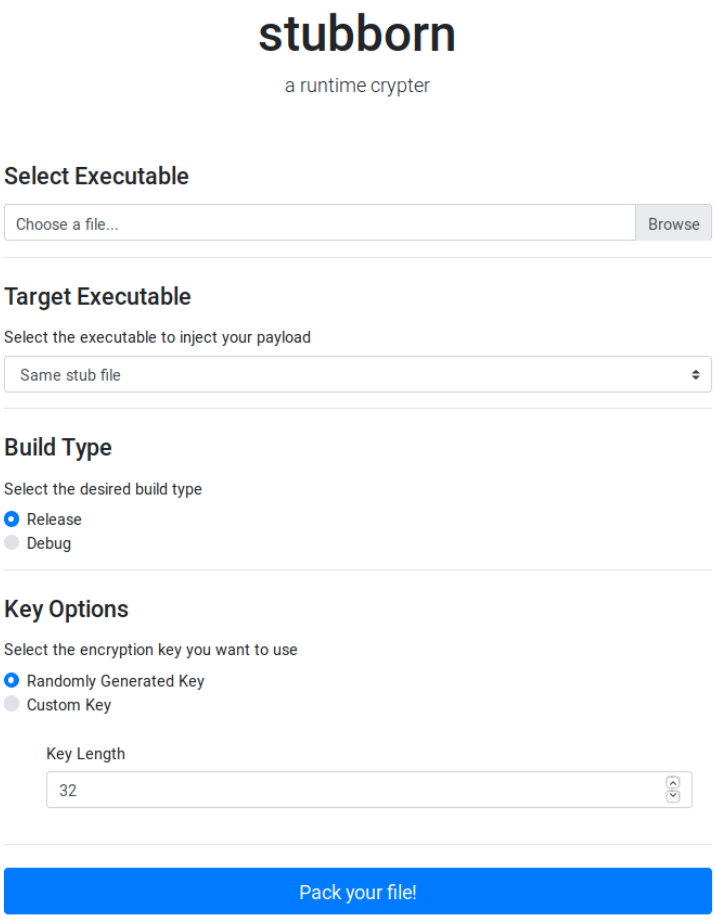


Figura 7.1: Formulari de la interfície d'usuari *crypter*.

en els arxius Python de l'aplicació que es troben dins la carpeta `app`. A continuació descriurem la funció que realitza cadascun d'aquests arxius.

L'aplicació Flask està definida en l'arxiu `app.py`. Només conté la ruta `/`, les peticions a la qual es gestionen mitjançant la funció `handle_form`. En cas que s'enviïn les dades del formulari, s'encarrega de validar-les, passar-les a la funció `build_stub`, i un cop aquesta funció retorna l'*stub*, enviar-lo a l'usuari per tal que se'l descarregui. En cas que es tracti d'una petició sense informació, o informació no vàlida, simplement es renderitzarà el formulari (mostrant els errors de validació que hi ha hagut, si és el cas).

La funció `build_stub` està definida dins l'arxiu `builder.py`. Resumint, aquest funció rep les dades del formulari, i s'encarrega de gestionar-les per crear l'*stub*. En primer lloc, xifra l'arxiu executable rebut mitjançant una clau (que pot ser proporcionada per l'usuari o bé creada de manera aleatòria). També s'encarrega de comprovar si aquest arxiu és de 32 bits (format PE32) o bé de 64 (PE32+) usant la llibreria `pefile` [5]. Finalment, amb

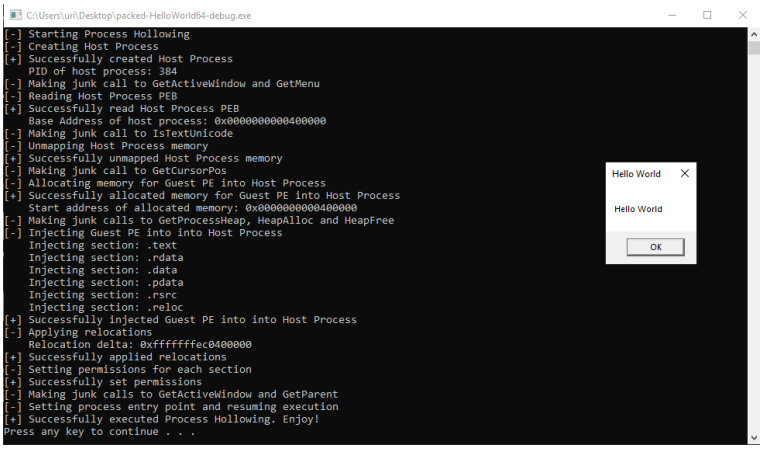


Figura 7.2: Exemple de l'execució d'un programa empaquetat amb el *crypter* en mode *debug*.

això i la resta d'informació que s'ha rebut del formulari, genera un diccionari que servirà per crear el projecte C++ a partir de la plantilla, mitjançant la llibreria `cookiecutter`. Aquest projecte es guarda a la carpeta `/tmp`, es compila usant les comandes `cmake` i `make`, i es retorna l'*stub* compilat.

Finalment, la resta d'arxius tenen un paper més secundari. Per una banda, a l'arxiu `encryption.py` hi trobem les funcions encarregades del xifratge, mentre que a l'arxiu `form.py` hi trobem la definició del model del formulari en la classe `StubbornForm`.

## 7.4 Creació del codi C++

El punt clau de la creació de l'*stub* és la creació del codi C++ a partir de la plantilla i la seva posterior compilació. Les plantilles estan contingudes dins de la carpeta `stub_templates`. Actualment, només hi trobem la plantilla per a l'*stub* de `Process Hollowing` que té la següent estructura:

```
process_hollowing/
├── {{ cookicutter.target_dir }}/
└── cookicutter.json
```

La carpeta `{{ cookicutter.target_project }}` és la que conté la plantilla del codi en sí (el nom segueix el sistema de plantilles de `cookicutter`), mentre que l'arxiu `cookicutter.json` ens permet definir certs camps que han de ser substituïts en la plantilla, així com els seus valors per defecte. Per exemple, si el contingut d'aquest arxiu conté:

```
{
 "target_dir": "some_path",
```

```
  "encryption_key": "supersecret",
  "key_length": 11,
  ...
}
```

`cookiecutter` substituirà qualsevol ocurrència del text `{{ cookicutter.target_dir }}` per `some_path`. Similarment, si troba el text `{{ cookicutter.encryption_key }}` ho substituirà per `supersecret`, etc. Ara bé, aquests són valors per defecte, però es poden canviar en el moment de crear la plantilla mitjançant un diccionari de Python que contingui aquestes claus. A continuació detallem tots els camps que estan definits en l'arxiu `cookicutter.json` i que, per tant, podem configurar abans de generar l'*stub*:

- `target_dir`: Directori on es guardarà el codi C++ i posteriorment es compilarà. Sempre és de la forma `/stubborn/tmp/<uuid>` on `uuid` és un identificador únic generat en el moment de la creació del projecte.
- `encryption_key`: La clau que s'ha de fer servir per desxifrar el *resource* (l'executable empaquetat dins l'*stub*).
- `key_length`: La llargada de la clau de xifratge.
- `target_exe_type`: Aquest paràmetre indica si s'ha d'injectar l'executable en un procés del propi *stub* o bé si és un altre executable (com `calc.exe`, etc.).
- `target_exe`: En el cas que sigui un altre executable, el nom d'aquest executable.
- `build_type`: El tipus de build (release o debug).
- `compiler`: El compilador que s'ha d'usar.
- `windres`: La versió de `windres` que s'ha d'usar (que serveix per copiar un *resource* dins un binari).
- `imports_key`: La clau amb la que s'han de desxifrar els noms de les importacions (vegeu Secció 6.1).
- `imports_key_length`: La llargada de la clau amb la que s'han de desxifrar els noms de les importacions.

Finalment, trobem un seguit de paràmetres del tipus `encrypted_<nom_dll>` o, similarment, `encrypted_<nom_importació>`, que simplement són els noms de les DLLs o importacions xifrats amb la clau especificada anteriorment (vegeu Secció 6.1).

Un exemple dels fitxers de codi generat es poden trobar en la carpeta `example/code` del repositori de `stubborn` [7]. Aquests arxius, ordenats per rellevància, són els següents:

1. `main.cpp`: Conté la part bàsica del codi: s'encarrega de carregar el *resource*, cridar les funcions per desxifrar-lo, executar el *Process Hollowing*, així com executar les funcions de detecció d'emulació de codi i *sandboxs*.

2. `processHollower.cpp` i `processHollower.h`: Codi i arxiu capçalera de la classe que executa la tècnica del *Process Hollowing* (vegeu el Capítol 5).

3. `hiddenImports.cpp` i `hiddenImports.h`: Codi i arxiu capçalera de les funcions per a l'ofuscació de les importacions (vegeu la Secció 6.1).

4. `junkApiCalls.cpp` i `junkApiCalls.h`: Codi i arxiu capçalera de les funcions encarregades de fer crides innòcues a l'API de Windows (vegeu les Secció 6.2).

5. `antivirusChecks.cpp` i `antivirusChecks.h`: Codi i arxiu capçalera de les funcions de detecció d'emulació de codi i *sandboxs* (vegeu les Seccions 6.3 i 6.4).

6. `decrypt.cpp` i `decrypt.h`: Codi i arxiu capçalera de les funcions encarregades del desxifratge del *resource*.

7. `windowsInternals.cpp` i `windowsInternals.h`: Codi i arxiu capçalera de les funcions per a gestionar estructures internes de Windows com el PEB, les capçaleres de l'executable, etc.

8. `resource.h` i `resource.rc`: Arxius per a la definició del *resource*.

9. `debug.cpp` i `debug.h`: Codi i arxiu capçalera de les funcions usades per a mostrar la informació de cada pas que es du a terme en mode *debug*.

A part d'aquests arxius de codi, també trobem l'arxiu `CMakeLists.txt`, que defineix el procés de compilació de l'executable, així com l'arxiu `crypt.exe`, que no és més que l'executable xifrat que s'haurà d'incloure com a *resource* dins de l'*stub*.

# Capítol 8

# Test de detecció d'antivirus

Un cop hem creat el *crypter*, podem passar a analitzar la seva eficàcia amb diverses eines antivirus per veure si realment hem aconseguit l'objectiu inicial o, si no, en quina mesura.

Per a fer-ho, hem usat la pàgina web VirusTotal [34]. Aquesta pàgina permet, de forma molt senzilla, analitzar arxius amb una setantena d'antivirus, obtenint de forma quasi immediata el resultat per a cadascun d'ells. Tot i que, com s'indica en la pàgina, és possible que alguns d'aquests antivirus no tinguin totes les funcionalitats que tenen les seves versions comercials, sí que ens permet obtenir una idea clara de si un arxiu pot ser considerat maliciós o no.

L'arxiu maliciós que hem escollit és un exemplar del Meterpreter (concretament una *reverse shell* TCP) de la plataforma de *penetration testing* Metasploit [26]. Meterpreter és de les eines més usades tant per a tests de penetració com per a actors maliciosos per obtenir accés de forma no autoritzada, i per tant hauria de ser detectat àmpliament per les eines antivirus. Efectivament, pujant l'arxiu a VirusTotal, el resultat obtingut (Figura 8.1) no deixa lloc a dubte.

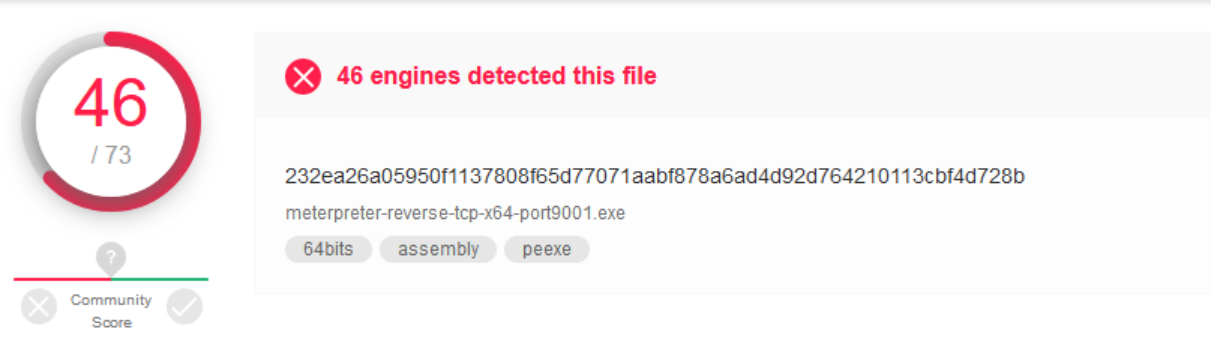


Figura 8.1: Resultat de VirusTotal de l'anàlisi de Meterpreter.

A continuació presentarem el resultat de dos tests diferents. En el primer analitzarem

un *stub* que només usi la tècnica de *Process Hollowing*, sense cap tècnica d'evasió d'antivirus, mentre que en el segon afegirem aquestes tècniques. Així doncs, veurem, si usant el nostre *crypter* aconseguim baixar el nombre de deteccions mostrades en la Figura 8.1 en dues situacions diferents.

Notem que aquests són els resultats en pujar per primer cop els arxius a VirusTotal. Aquesta plataforma pot distribuir els arxius a professionals de l'anàlisi de *malware* i empreses antivirus, i per tant és possible que en un futur aquests mateixos arxius siguin detectats mitjançant signatures per més eines de les que es mostren a continuació.

## 8.1 Resultats del primer test

En aquesta secció mostrarem els resultats de l'anàlisi d'un exemplar del Meterpreter empaquetat amb el nostre *crypter* sense usar cap de les tècniques d'evasió d'antivirus presentades en el Capítol 6.

Notem que actualment, la majoria d'antivirus no es limiten només a buscar signatures a l'hora d'analitzar un arxiu, si no que també es basen en regles heurístiques [19]. Aquestes regles consisteixen en detectar codi maliciós analitzant el codi desensamblat, inspeccionant les diferents crides que es fan a diferents funcions, i inclús poden executar l'arxiu en un entorn emulat o instal·lant *hooks* a diferents funcions de l'API de Windows.

Per aquest fet, doncs, i tenint en compte que les tècniques usades per a la creació del *crypter* són àmpliament conegudes per la comunitat que es dedica a l'anàlisi de *malware*, el primer que podem pensar és que el resultat d'aquest primer test no distarà significativament del resultat mostrat a la Figura 8.1. No obstant, la diferència obtinguda és bastant substancial (Figura 8.2): només 4 *scanners* han classificat l'arxiu com a maliciós enfront de les 46 del cas anterior. Notem, a més, que només una de les eines (Microsoft), etiqueta correctament l'arxiu com a Meterpreter.

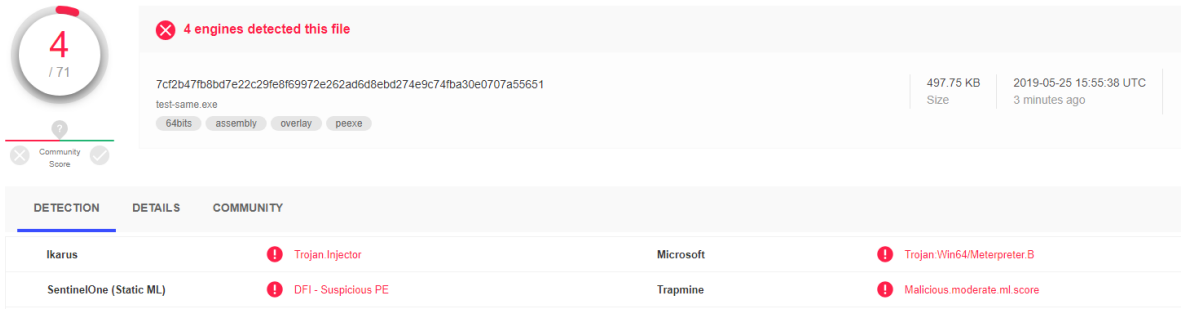


Figura 8.2: Resultat de VirusTotal de l'anàlisi de Meterpreter empaquetat sense tècniques d'evasió d'antivirus.

Sembla, doncs, que les tècniques usades han fet el seu efecte. A continuació, però, veurem si podem encara millorar una mica aquest resultat.

## 8.2 Resultats del segon test

En aquesta secció mostrarem els resultats de l'anàlisi d'un exemplar del Meterpreter empaquetat amb el nostre *crypter*, aquest cop usant les tècniques descrites en el Capítol 6: ofuscació d'importacions, crides a l'API de Windows falses, detecció d'emulació de codi i detecció de *sandbox*. És interessant saber quina efectivitat tenen aquestes tècniques, o si inclús poden tenir un efecte contrari a l'esperat.

El resultat obtingut es pot veure en la Figura 8.3. Veiem doncs, que hem millorat el

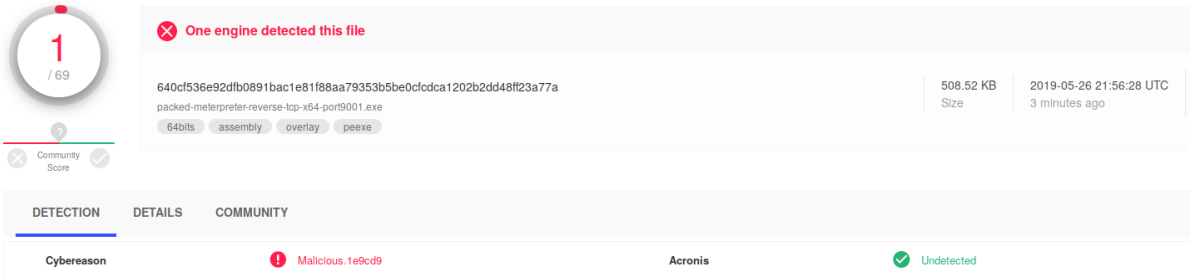


Figura 8.3: Resultat de VirusTotal de l'anàlisi de Meterpreter empaquetat amb tècniques d'evasió d'antivirus.

resultat de la secció anterior, tot i que encara hem obtingut una detecció. És curiós, però, que l'eina que ha detectat aquest arxiu no havia detectat l'anterior (vegeu la Figura 8.2). No obstant, aquest resultat ens fa pensar que possiblement aplicant alguna tècnica més podríem aconseguir un resultat de zero deteccions.

# Capítol 9

# Conclusions i treball futur

D'aquest treball en podem extreure vàries conclusions. En primer lloc, el fet de desenvolupar un *crypter* ens ha permès entendre amb profunditat el format PE, familiaritzar-nos amb l'API de Windows, així com aprendre tècniques que usa freqüentment el *malware* com són la tècnica de *Process Hollowing* o les diferents tècniques d'evasió d'antivirus que hem implementat. Aquest aprenentatge per si sol ja és valuós de cara a entendre millor els reptes de seguretat als que s'enfronten els sistemes informàtics actuals.

D'altra banda, però, hem pogut comprovar de primera mà la relativa facilitat amb què es pot crear un *crypter* que, tot i que no em aconseguit que fos FUD, no sigui detectat per pràcticament cap antivirus. Notem que les tècniques usades són accessibles per a qualsevol persona que tingui interès en buscar-les. Això també ens permet entendre fins a quin punt un antivirus pot protegir-nos, i la necessitat d'altres eines que puguin protegir l'espai que els antivirus no cobreixen. En aquest sentit, doncs, cal destacar la importància de la protecció en temps real o de les eines d'*Endpoint Detection and Response* (EDR), que van més enllà de l'anàlisi que pot fer un antivirus.

Finalment, també podem concloure que el resultat d'aquest treball, el *crypter* que hem anomenat `stubborn`, és una eina que pot ser útil de cara al futur. Efectivament, pot ajudar a fer proves en els propis sistemes per comprovar l'eficàcia d'eines antivirus o per a realitzar tests de penetració. A més, el seu desenvolupament pot continuar en un futur, afegint funcionalitats, millorant certs aspectes, etc.

En aquest sentit, doncs, hi ha moltes tasques que poden seguir després de la realització d'aquest treball. Per una banda, es pot millorar la interfície d'usuari, permetent més opcions de configuració, fent-la més atractiva, etc. També seria molt interessant implementar altres tècniques d'injecció de processos i comparar els resultats de detecció d'antivirus amb els que hem obtingut en aquest treball. De la mateixa manera, es poden introduir moltes més tècniques d'evasió d'antivirus, i també incloure-hi tècniques d'*antidebugging*, etc. Un altre aspecte que es podria ampliar és la disponibilitat d'algorismes de xifratge: implementar el xifrat amb AES, per exemple, que és més robust que l'algorisme XOR que hem usat. Finalment, un altre projecte que seria molt interessant seria donar suport a

binaris de Linux. En aquest cas, caldria entendre bé el format ELF i buscar tècniques d'injecció de processos específiques per a Linux.

En resum, aquest treball ens ha servit per crear un *crypter* que podem usar com a base per seguir aprenent, investigant i desenvolupant per tal d'entendre millor el món complex de la seguretat informàtica en general i el funcionament del *malware* en particular.

# Referències


[1] Christian Ammann. *Hyperion: Implementation of a PE-Crypter*. 2012.

[2] Bill Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system*. Ed. de Jones & Bartlett Publishers. 2a ed. 2012. 346 - 347.

[3] Tom Brosch i Maik Morgenstern. *Runtime packers: The hidden problem*. 2006.

[4] Ero Carrera. *PE Header Walkthrough*. URL: `https://drive.google.com/file/d/0B3_wGJkuWLytQmc2di0wajB1Xzg/view`.

[5] Ero Carrera. *pefile*. URL: `https://github.com/erocarrera/pefile`.

[6] Ero Carrera. *Portable Executable Format Layout*. URL: `https://drive.google.com/file/d/0B3_wGJkuWLytbnIxY1J5WUs4MEk/view`.

[7] Oriol Castejón. *Stubborn, a runtime crypter*. 2019. URL: `https://github.com/ocastejon/stubborn`.

[8] *CMake*. URL: `https://cmake.org/`.

[9] *Cookiecutter*. URL: `https://cookiecutter.readthedocs.io/en/latest/readme.html`.

[10] *core-packer (codi filtrat)*. URL: `https://github.com/hackedteam/core-packer`.

[11] Will Cummings i Ethan Heilman. *A Brief Examination of Hacking Team's Crypter: core-packer*. 2015. URL: `http://ethanheilman.tumblr.com/post/128708937890/a-brief-examination-of-hacking-teams-crypter`.

[12] Mike Czumak. *peCloak.py – An Experiment in AV Evasion*. 2015. URL: `https://www.securitysift.com/pecloak-py-an-experiment-in-av-evasion/`.

[13] *Docker*. URL: `https://www.docker.com/`.

[14] *Flask*. URL: `http://flask.pocoo.org/`.

[15] *flat assembler (fasm)*. URL: `https://flatassembler.net/`.

[16] Max Goncharov. *Russian underground 101*. 2012.

[17] hasherezade. *RunPE*. URL: `https://github.com/hasherezade/libpeconv/tree/master/run_pe`.

[18] *Jinja2*. URL: http://jinja.pocoo.org/.

[19] Elias Bachaalany Joxean Koret. *The Antivirus Hacker's Handbook*. John Wiley & Sons, 2015.

[20] Rémi Jullian. *In-depth Formbook malware analysis – Obfuscation and process injection*. URL: https://thisissecurity.stormshield.com/2018/03/29/in-depth-formbook-malware-analysis-obfuscation-and-process-injection/.

[21] John Kennedy et al. *PE Format*. 2019. URL: https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format.

[22] Brian Krebs. *Criminal Classifieds: Malware Writers Wanted*. 2011. URL: https://krebsonsecurity.com/2011/06/criminal-classifieds-malware-writers-wanted/.

[23] John Leitch. *Process Hollowing*. URL: http://www.autosectools.com/Process-Hollowing.pdf.

[24] Robert Lyda i James Hamrock. *Using entropy analysis to find encrypted and packed malware*. 2007.

[25] Fernando Mercês. *The Brazilian underground market*. Inf. tèc. Trend Micro, 2014, pàg. 8. URL: https://www.trendmicro.de/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-brazilian-underground-market.pdf.

[26] *Metasploit Framework*. URL: https://www.metasploit.com/.

[27] *MinGW, Minimalist GNU for Windows*. URL: http://www.mingw.org/.

[28] Alberto Ortega. Ed. de Pafish. URL: https://github.com/a0rtega/pafish.

[29] *PEB structure*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/api/winternl/ns-winternl-_peb.

[30] Matt Pietrek. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format*. 1998. URL: https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10).

[31] Johannes Plachy. *Protable Executable File Format*. 1997. URL: http://www.skynet.ie/~caolan/pub/winresdump/winresdump/doc/pefile.html.

[32] Adrian Stepan. *Improving proactive detection of packed malware*. 2006.

[33] Arne Swinnen i Alaeddine Mesbahi. *One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques*. 2014.

[34] *VirusTotal*. URL: https://www.virustotal.com.

[35] Jos Wetzels. *The flawed crypto of Hacking Team's 'core-packer' malware crypter*. 2015. URL: https://samvartaka.github.io/malware/2015/09/13/hackingteam-crypter.

[36] *Windows API resolution via hashing*. URL: https://github.com/LloydLabs/Windows-API-Hashing.

[37] *Yoda's Protector*. URL: http://yodap.sourceforge.net/.