Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

TÍTOL: Ofuscament de malware per al bypass dels antivirus comercials

AUTOR: OLIVÉ MONTEJO, PAU

DATA DE PRESENTACIÓ: Febrer, 2021

COGNOMS: Olivé Montejo **NOM: Pau**

TITULACIÓ: Grau en Enginyeria Informàtica

PLA: 2018

DIRECTOR: Neus Català i Roig

DEPARTAMENT: Ciències de la Computació

QUALIFICACIÓ DEL TFG

TRIBUNAL

PRESIDENT	SECRETARI	VOCAL
Josep Maria Merenciano Saladrigues	Jose Antonio Roman Jimenez	Jose Ramon Piney Da Silva

DATA DE LECTURA: 4 de febrer de 2021

Aquest Projecte té en compte aspectes mediambientals: Sí X No

RESUM

En aquest treball es proposa implementar un *crypter* (o *Software Packer*) com a *Proof of Concept* i detallar quines tècniques d'ofuscació es poden utilitzar per convertir un executable *malware* actualment detectable, en un executable indetectable tant en temps d'escaneig (*Scan time*) com en temps d'execució (*Run time*) a ulls dels antivirus comercials.

El *crypter* s'engloba en tres grans blocs: encriptació del *malware*, generació d'un *stub* perquè es pugui desencriptar a ell mateix i càrrega de l'executable desencriptat directament a memòria sense tocar el disc dur.

Aquest treball redacta en detall quin és el disseny seguit per a l'implementació del nostre *crypter* i quines decisions s'han pres. Per poder implementar la part de l'*stub* del *crypter*, serà necessari utilitzar un mètode d'injecció de codi dinàmicament a memòria. En aquest cas s'ha escollit el mètode conegut com a *Process Hollowing*. Addicionalment, a l'annex, hi apareixen altres alternatives documentades de manera més genèrica.

De manera resumida, el *Process Hollowing* és una tècnica que es basa en: *a*) crear un nou procés en estat suspès; *b*) modificar-ne el contingut amb la imatge d'un altre executable (en aquest cas, els bytes ja desencriptats que contenen el *malware*); *c*) realitzar les modificacions necessàries per a la correcta execució del nou procés, i *d*) reprendre l'execució d'aquest nou procés que ara conté els bytes del *malware* en el seu espai de memòria.

En aquest projecte, es documentaran i s'aplicaran tècniques d'ofuscació addicionals al *crypter* per millorar els resultats obtinguts, és a dir, per reduir les ràtios de detecció. Algunes d'aquestes tècniques són: ofuscació d'importacions, clonatge de certificat digital o anti entorns d'emulació, entre d'altres.

Per valorar l'eficàcia del *crypter* creat, es realitzaran uns tests a partir d'una servei web que actua com a *multiscanner* d'antivirus. Aquesta servei online ofereix la possibilitat d'analitzar qualsevol arxiu executable amb els 26 antivirus comercials de més renom, obtenint els resultats gairebé instantàniament.

El propòsit d'aquest treball és reduir les ràtios de detecció de qualsevol *malware* al mínim possible, sent el millor resultat un arxiu *malware FUD* (*Fully Undetectable*), utilitzant el *crypter* amb tècniques d'ofuscació addicionals.

Paraules clau (màxim 10):

Crypter	Process Hollowing	WinAPI	C++
Antivirus	Malware	Multiscanner	Software Packer
FUD	Ofuscació		

ABSTRACT

This project aims to implement a *crypter* (also known as *Software Packer*) as a Proof of Concept and to detail which obfuscation techniques can be used to convert a currently detected malware executable into an undetectable executable both in scan time and run time towards commercial antivirus.

The crypter consists of three main blocks: encryption of the malware, creation of a stub with the ability to decrypt itself and self-loading of the decrypted malware directly into memory without touching the hard drive.

This report describes in detail what are the design patterns followed for the implementation of our crypter and what decisions have been taken. To be able to implement the stub it is necessary to choose between several code injection methods. In our case, the *Process Hollowing* method has been chosen. You can find other code injection methods briefly explained in the appendix.

In short, *Process Hollowing* is a technique based on: a) creating a new process in a suspended state, b) modifying the content with the image of another executable (in this case, for the already decrypted bytes containing the malware), c) making the necessary modifications for the correct execution of the new process, d) resuming the execution of this new process which now contains the malware bytes in its memory space.

In this project, additional obfuscation techniques to the crypter will be used and documented to improve the results obtained from it. Some of these techniques are the following: obfuscation of imports, digital certificate cloning or anti-emulation, among others.

In order to evaluate the effectiveness of the crypter created, tests will be carried out using a website that acts as an antivirus multiscanner. This website offers the possibility of scanning any executable file with the 26 most renowned commercial antiviruses, obtaining the results almost instantly.

The purpose of this work is to reduce the detection rates of any malware to the lowest possible, the best result being a *FUD* (Fully Undetectable) malware file, by using the crypter with additional obfuscation techniques.

Keywords (10 maximum):

Crypter	Process Hollowing	WinAPI	C++
Antivirus	Malware	Multiscanner	Software Packer
FUD	Obfuscation		

SUMARI

Sumari de figures

Sumari de taules

Glossari

- **Signatura de virus**: Conjunt únic de bytes que identifiquen un virus en concret.

- **Ofuscar**: Dificultar l'enteniment de la funcionalitat i de les intencions d'un executable.

- ***Malware***: Arxiu executable o DLL amb intencions malicioses.

- ***Payload***: Peça clau del *malware*. És la part del codi que realment conté la part funcional d'aquest.

- ***Crypter / Software Packer***: Eina utilitzada per ofuscar els arxius *malware* de cara a la protecció dels antivirus.

- **FUD**: Acrònim anglès de *Fully Undetectable*. Fa referència al fet que un *malware* sigui completament indetectable davant de tots els antivirus comercials.

- **PE**: Acrònim anglès de *Portable Executable*. Format específic de Microsoft que segueixen els executables i les DLL en el sistema operatiu de Windows.

- **WinAPI**: Biblioteca per als desenvolupadors d'aplicacions que s'executaran en el sistema operatiu de Windows.

- **TLS:** Acrònim anglès de *Thread Local Storage*. És un mètode utilitzat perquè els subprocessos d'un procés amb *multithread* puguin emmagatzemar dades locals.

- **Fòrum *underground***: Lloc de discussió on els usuaris intercanvien informació i ofereixen serveis relacionats amb el món del *hacking*.

- ***Black Hat***: Part del *hacking* on es deixa de banda l'ètica i la moral per al benefici personal o, en alguns casos, diversió.

- ***Bypass***: Mètode que aconsegueix esquivar un o varis mecanismes de defensa i protecció relacionats amb el software.

- **Polimorfisme**: Tècnica utilitzada en el *malware* que permet mutar el codi de l'executable mateix en cada execució, mantenint la mateixa funcionalitat.

- **MS-DOS**: Un dels sistemes operatius de la familia DOS per part de Microsoft en els anys 1980.

- ***Offset***: Nombre enter que indica la distància o desplaçament des d'un punt inicial fins a un element en concret.

- ***Malware spreading***: Fet de propagar el contagi del *malware* a més víctimes.

- **GUI**: Acrònim anglès de *Graphical User Interface*. Interfície gràfica que facilita l'ús d'una aplicació mitjançant imatges i altres representacions gràfiques.

- ***Refactoring***: Reescriure la mateixa part d'un codi de diferent manera però mantenint la mateixa funcionalitat.

- **VPN**: Acrònim anglès de *Virtual Private Network*. Xarxa privada que permet una extensió de la xarxa local sobre una xarxa pública o no controlada d'Internet. L'origen del tràfic de l'usuari que es connecta serà diferent que el proporcionat pel proveïdor d'Internet.

- **API**: Acrònim anglès de *Application Programming Interface*. Funcions, mètodes i processos que pot oferir una determinada biblioteca o servidor.

- ***Webapp***: Aplicació software que corre en un servidor i és consumida des d'un navegador web.

- ***Endpoint***: Punt final de comunicació de l'API d'un servidor que pot ser consumit des d'un altre sistema. Pot ser, per exemple, una URL.

- ***Roadmap***: Planificació temporal de tasques a dur a terme per a arribar a un objectiu específic.

- **IDE**: Acrònim anglès de *Integrated Development Environment*. Eina informàtica per al desenvolupament de programari de manera còmoda i ràpida.

- **CMD**: Acrònim anglès de *Command Prompt (Microsoft Windows)*. Intèrpret d'ordres de Windows que permet als usuaris interactuar amb el sistema operatiu.

- ***Spinner***: Element web que es sol utilitzar per indicar a l'usuari que un recurs està sent carregat.

1. Introducció i contextualització

1.1 Context

Els antivirus són la primera línia de prevenció i protecció de tots els sistemes informàtics moderns.

Dins del món del programari maliciós (*malware*), des de l'aparició dels primers antivirus comercials, hi ha una guerra constant entre aquestes empreses i els desenvolupadors de *malware*. L'objectiu d'aquests últims és aconseguir que els seus *payloads* siguin totalment indetectables davant de qualsevol sistema de seguretat. Una de les maneres d'aconseguir-ho és amb l'ús d'una eina anomenada *Crypter*, *Software Packer* o *Packer*, que no deixen de ser programes que encripten un executable (normalment maliciós i actualment detectat per molts antivirus), i hi afegeixen funcions úniques encarregades de desencriptar aquest mateix executable inicial només quan l'usuari executi l'arxiu. Un cop aquest arxiu sigui executat, serà desencriptat en temps d'execució i carregat directament a memòria, evitant així la detecció estàtica dels antivirus que es basen en la comprovació de *signatures de virus*. Els millors *crypters* són aquells que aconsegueixen ser *FUD* (*Fully Undetectable*), és a dir, que són aquells capaços de tornar qualsevol *malware* indetectable davant de tots els antivirus comercials existents.

Per a que un arxiu maliciós no sigui detectat pels antivirus, no és explícitament necessari l'ús d'un *crypter,* doncs un desenvolupador pot implementar les seves pròpies tècniques d*'ofuscació* dins del mateix arxiu binari. Però l'avantatge que otorga l'ús d'un *crypter* és que el fet de separar el *malware* de l'eina que l'ofuscarà, permet a l'usuari reutilitzar-lo per a tot tipus d'executables, inclús aquells els quals no han estat desenvolupats per la mateixa persona que està utilitzant el *Packer*.

També és important saber que l'ús d'aquestes eines d'ofuscació no té com a únic objectiu el cibercrim, prevenint la detecció dels virus davant els sistemes de detecció dels antivirus. El fet que un arxiu binari estigui encriptat, també vol dir que a aquest serà molt difícil d'aplicar-li l'enginyeria inversa. Moltes empreses volen mantenir en secret certes parts del seu codi font, sobretot en el món dels videojocs, ja què això evita a

molts possibles tramposos [1], o també en productes comercials per evitar ser piratejats amb facilitat [2].

L'objectiu d'aquest treball és l'estudi de les tècniques d'ofuscació que s'utilitzen en els *crypters* moderns i la implementació d'un *crypter* com a *Proof of Concept* (PoC), posant a prova la majoria d'antivirus comercials. Això suposarà l'estudi de les tècniques de detecció i funcionament que utilitzen aquests antivirus. També haurem d'aprofundir en el funcionament intern de Windows, doncs serà necessari tenir una idea ben clara sobre com funcionen i quina estructura mantenen els *PE* (*Portable Executable*) d'aquest sistema operatiu i haurem d'utilitzar funcions de la pròpia API de Windows (*WinAPI*).

1.2 Introducció als *crypters*

En aquest apartat s'intentarà aprofundir una mica més dins del concepte dels *crypters* o *packers*.

Hi ha moltes maneres d'implementar un *crypter* i aquests poden estar programats amb diferents llenguatges de programació.

Com s'ha mencionat a l'apartat anterior, un *crypter* és un programari encarregat d'agafar qualsevol executable (ja sigui programari maliciós o no), llegir byte a byte el seu contingut i encriptar-lo en una memòria intermèdia o *buffer*, per tal que aquest buffer sigui posteriorment desencriptat en temps real d'execució i carregat dinàmicament a memòria sense tocar el disc.

Un *crypter* està format principalment per dos projectes:

- ***Builder*:** Aquest projecte serà l'encarregat de llegir i encriptar l'executable que volem ofuscar. L'executable encriptat serà emmagatzemat en un buffer i, a partir d'aquí, hi ha dues maneres principals d'afegir aquest buffer a l'altre projecte, el de l'*stub*. La primera manera és la d'emmagatzemar aquest buffer com a sortida en un arxiu capçalera que serà utilitzat en el segon projecte (el de l'*stub*). Per exemple, emmagatzemar-lo a l'arxiu "`shellcode.h`". L'altra manera és la de simplement guardar aquest buffer com si fos un

recurs dins del segon projecte (el mateix que es faria per agregar una icona, per exemple).

- ***Stub*:** Aquesta és la part més important del *crypter*, ja que serà el projecte encarregat de llegir el buffer amb l'executable encriptat, desencriptar-lo i carregar-lo dinàmicament a memòria. També s'aplicaran mètodes addicionals per reduir les possibles deteccions dels antivirus.
  Per carregar un executable a memòria hi ha varis mètodes d'injecció que es poden utilitzar. La injecció de codi a memòria consisteix bàsicament en carregar els bytes d'una imatge binària dins d'un procés actiu qualsevol, de tal manera que aquest procés es passi a comportar com l'executable que formaven aquests bytes però sense haver-lo executat directament des del disc dur. En aquest treball s'explicarà detalladament el mètode d'injecció utilitzat i a l'annex es podrà trobar informació d'altres mètodes existents.

1.3 Organització del treball

Aquest treball, per tant, té la intenció de cobrir tant la part teòrica del funcionament dels antivirus davant d'amenaces *malware*, l'estructura interna dels executables de Windows i alguns dels possibles mètodes d'injecció, com la part pràctica amb el desenvolupament d'un *crypter* com a *Proof of Concept*.

En quant a la part pràctica, aquest *crypter* constarà d'un total de 4 projectes:

1. ***PEEncrypter***
2. ***PELoader***

} *Builder* i *Stub* del *crypter*, respectivament. Ja explicats breument en l'apartat anterior.

3. ***SignatureClone***: Copia el certificat digital d'un executable reconegut a qualsevol altre executable. És una de les tècniques utilitzades per reduir les ràtios de detecció.

4. ***WebApp***: Projecte de presentació per unificar la resta de projectes en una sola pàgina web donant a l'usuari l'opció de penjar

l'executable que vol encriptar, la clau utilitzada per encriptar i desencriptar els binaris, la icona de l'executable resultant, l'executable del que vol copiar el certificat digital (opcional) i el nom del fitxer de sortida.

1.4 Stakeholders

Aquest treball està dirigit a aquelles persones interessades en el món de la ciberseguretat i, especialment, pel *malware* que té com a objectiu atacar a màquines que utilitzen el sistema operatiu Windows. Per assolir aquest objectiu, han de conèixer quines són algunes de les tècniques emprades per fer front a la defensa dels antivirus coneguts, ja sigui per motius d'adquirir coneixement i context, o perquè tenen intenció de fer enginyeria inversa a programari que apliqui algunes de les tècniques aquí descrites en el futur.

També pot resultar interessant per a aquells estudiants universitaris que s'hagin quedat amb la curiositat de veure com funcionen els arxius binaris i el maneig i context dels processos en Windows, després d'haver estudiat algunes de les crides de sistema i la gestió de processos de Linux en l'assignatura de Sistemes Operatius.

Pot ser intrigant per aquelles persones amb cert gust cap a la programació de baix nivell i aquelles interessades en introduir-se en les definicions de tipus (*data types*) i determinades funcions de l'API de Windows, especialment en C++.

També pot resultar d'ajuda a aquelles entitats que tinguin intenció de llançar un producte software propi de manera més professional i que utilitzin aquestes tecnologies amb la intenció de comercialitzar-lo com a objectiu d'oferir protecció contra l'enginyeria inversa, no fent-se responsable de l'ús que li donin els seus usuaris.

1.5 Antivirus

Els antivirus comercials són el software més utilitzat de cara a la protecció de dades i equips informàtics, sobretot pels usuaris comuns però també a nivell d'empreses.

En general, les llicències per empreses no són gratuïtes i això els provoca costos afegits. A més, els antivirus tampoc garanteixen una protecció al 100%, perquè la majoria usen tècniques de detecció basades en signatures que són vulnerables als virus que no hagin estat detectats anteriorment.

Com que es podria dedicar un treball sencer al funcionament dels antivirus, ja que són programes molt complexos, i l'objectiu d'aquest treball no està centrat explícitament en aquests sinó en entendre les seves estratègies per la detecció de *malware*, en aquest apartat es tractaran els aspectes necessaris que s'han de saber sobre la manera en la que detecten arxius maliciosos per així tenir un coneixement general dels passos que s'han de seguir per poder fer un *bypass* als seus sistemes de detecció, sense entrar gaire en detall.

1.5.1 Tècniques utilitzades per a la detecció de *malware*

Els mètodes més rellevants que hem de conèixer per al nostre objectiu són:

- **Detecció basada en signatures:** Aquesta tècnica, principalment, té a veure amb l'anàlisi manual feta pels investigadors de *malware* (tot i que solen utilitzar eines que els automatitzen certes tasques) amb l'objectiu de trobar patrons en els arxius maliciosos. Quan un arxiu és detectat com a maliciós, es generen noves signatures relacionades i s'emmagatzemen a les seves bases de dades. Més endavant, quan un antivirus realitza un escaneig, utilitza tècniques de concordança de patrons (*pattern-matching*) i compara les possibles signatures detectades amb les de la seva base de dades. Hi ha molts tipus possibles de signatures i inclús algunes poden arribar a generar falsos positius. Aquesta tècnica no és gaire efectiva contra nous *malwares* desconeguts i, fins i tot, amb molts *malwares* ja coneguts

tampoc acaba de ser molt efectiva, ja que poden arribar a utilitzar *polimorfisme*[1] en el seu codi mateix.

- **Heurístiques:** És una de les tècniques que s'utilitza per detectar virus nous o poc distribuïts dels quals encara no es posseeix cap signatura a la base de dades relacionada amb el seu codi maliciós. La idea general és la d'utilitzar definicions genèriques de *malware* (que no deixen de ser un tipus de signatures però de caràcter general) per poder detectar variants d'aquests. Al distribuir els diferents *malwares* existents en famílies, fa possible reconèixer àrees úniques que permeten crear signatures genèriques.

  Alguns dels mètodes heurístics utilitzen tecnologies relacionades amb la Intel·ligència Artificial, de tal manera que proporcionen als antivirus maneres intel·ligents de detectar el *malware* automàticament i analitzar el codi realitzant una inspecció profunda de la seqüència d'instruccions amb els virus desconeguts [3].

- **Detecció en temps real d'execució:** Aquest mètode de detecció està basat en l'anàlisi de comportament del *malware* mentre aquest està sent executat. Bàsicament, l'antivirus monitoritza tots els processos en execució. Això li permet investigar els processos que s'estan comportant d'una manera inusual, com per exemple aquells que volen connectar-se a un servidor extern remot i descarregar arxius. Aquest mètode, dins de la categoria dels antivirus, pot rebre el nom de *real-time scanners*, *residents* i *on-access scanners*.

  "*The resident analyzes files that are accessed, created, modified, or executed by the operating system or other programs (like web browsers); it does this to prevent the infection of document and program files by viruses or to prevent known malware files from executing.*" [4, Pàg. 8]

- ***Sandboxing*:** Aquest mecanisme consisteix en executar els programes en un entorn separat / aïllat controlant tots els recursos assignats en cas de danys. Es considera un mecanisme de contenció contra les tècniques d'ofuscació del *malware*, ja que realment no es

[1] El poliformisme utilitzat en els *crypters*, és una tècnica molt avançada que permet canviar la informació binària de l'executable resultant mantenint la mateixa funcionalitat intacta de tal manera que aquest sigui inidentificable durant l'anàlisi estàtica de signatures. A ulls del antivirus és com si es generessin instàncies del mateix *malware* aparentment diferents.

pot saber quin serà el comportament d'un arxiu ofuscat fins que aquest sigui executat.

Hi ha diverses eines que fan servir aquesta tècnica. En general, aquestes eines, poden imitar la interacció d'un arxiu maliciós, així com detectar i documentar els canvis que s'han provocat en un sistema infectat. Així doncs, un cop s'emula l'execució d'una mostra del *malware*, és important recollir-ne tanta informació com sigui possible per determinar els canvis realitzats al sistema, identificar patrons i comprendre'n el comportament.

1.6 Format dels *Portable Executable*

Abans d'entrar en detall amb el funcionament dels *crypters* és important entendre com s'estructura un fitxer executable en Windows.

Un *Portable Executable* (PE) és el format estàndard utilitzat per Microsoft per als fitxers executables i les biblioteques d'enllaç dinànamic (DLL), entre d'altres. Va ser inicialment introduït amb el sistema Windows NT i està basat en l'especificació del format COFF (*Common Object File Format*) usat en Unix.

És important tenir un coneixement bàsic de l'estructura d'aquest format sobretot perquè una part fonamental dels *crypters* és que l'executable resultant d'aquests ha de tenir la capacitat de desencriptar-se a ell mateix i carregar la part desencriptada de l'arxiu directament a memòria sense tocar el disc. Per poder fer això, haurem de saber recórrer l'executable ja desencriptat seguint les definicions d'aquest format i poder anar escrivint els seus continguts en el nou espai de memòria assignat.

El format PE és una estructura de dades que proporciona la informació necessària perquè el *loader*[2] de Windows pugui gestionar el codi de l'executable de la millor manera possible. Això inclou poder resoldre referències a DLL, taules d'importació i exportació de les API, una gestió eficient de recursos i de dades d'emmagatzematge local de subprocessos (*TLS*).

[2] El *loader* de Windows és l'encarregat de carregar els arxius executables a memòria. Per fer-ho, crea un espai d'adreces virtual i assigna el mòdul de l'executable des del disc dur a aquest espai. Llavors, gestiona totes les inicialitzacions necessàries perquè aquest procés pugui ser executat correctament (com la pila, el *thread* principal, el context de l'executable, càrrega de les DLL necessàries, etc.).

En la Figura 1.1 podem veure una representació gràfica del format PE.

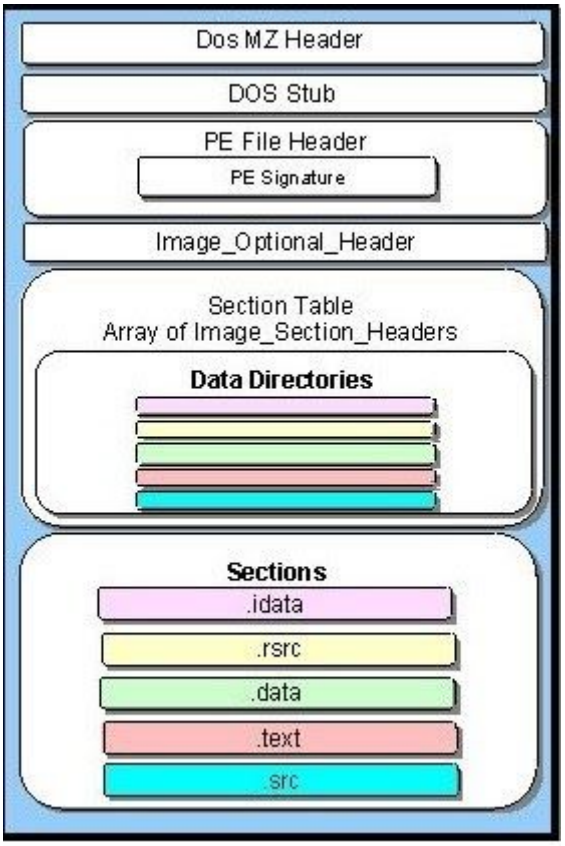

Figura 1.1 Estructura interna d'un fitxer PE.

Estructura del PE

En aquest apartat es descriuen les parts més rellevants del format PE. Són les parts que més influència tindran de cara al desenvolupament del *crypter*. Si bé alguns dels camps que es descriuran a continuació dins de cada bloc no seran utilitzats, considero que és important descriure la seva funcionalitat per així ajudar a consolidar la idea general i les possibilitats que otorga el coneixement d'aquest tipus d'estructura de fitxer.

Al cap i a la fi, dins del *crypter* a implementar, el que caldrà fer serà recórrer les estructures internes de l'executable i anar fent les modificacions necessàries. Més endavant ho veurem en detall.

1.6.1 El *DOS Header* i el *DOS Stub*

Les dues primeres seccions inicials d'un PE existeixen com a mode de compatibilitat cap enrere, perquè així qualsevol executable pugui ser executat també en MS-DOS.

La primera secció és la ***DOS Header*** de la qual, bàsicament, els únics camps que ens interessen són: *a*) el camp `e_magic`, que no és res més que la signatura que l'identifica com un arxiu compatible amb MS-DOS (sempre amb el valor 0x54AD) i es troba a l'*offset* inicial 0x0, i *b*) el camp `e_lfanew`, que ens indica la posició relativa necessària perquè el *loader* de Windows es salti l'*stub* DOS (que veurem a continuació) i vagi directament a la capçalera PE. Aquest camp es troba a l'*offset 0x3C* i ens serà realment important a l'hora d'accedir, més endavant, a les parts essencials dels nostres PE.

La segona secció és l'anomenada ***DOS Stub***, on bàsicament hi ha un petit programa que mostra un missatge d'error de l'estil: "*This program cannot be run in DOS mode*", en el cas que aquest arxiu fos executat en MS-DOS. Com he comentat anteriorment, aquestes capçaleres existeixen amb el propòsit de compatibilitat cap enrere en ment.

1.6.2 Les capçaleres PE

Els primers quatre bytes que trobem a la capçalera PE són els de la seva signatura, amb el valor de `50 45 00 00` que es tradueix a "PE" seguit de dos *nulls* (o zeros). És important recordar i remarcar que a aquesta posició podrem accedir-hi gràcies al camp descrit anteriorment en el *DOS Header* anomenat `e_lfanew`.

En aquesta capçalera podem trobar informació general de l'arxiu que s'utilitza per indicar-li al sistema com tractar-lo, doncs hi ha camps com el `Machine` que indiquen per a quin tipus de processador va ser construït l'executable, o el camp `Characteristics`, que proporciona informació del tipus d'arxiu (.exe, .dll, mode debug) i altres tipus d'informació com ara si l'arxiu conté una taula de relocalitzacions (ho veurem més endavant).

Realment, els únics camps rellevants aquí, a part de la signatura que ens pot servir per fer comprovacions al nostre codi de si l'arxiu amb el que estem tractant és un arxiu PE vàlid, són els camps `NumberOfSections`, que més endavant veurem la seva importància, i `SizeOfOptionalHeader`, que és la mida de la següent capçalera i ens pot servir també com a *offset* relatiu per desplaçar-nos dins del fitxer PE. També farem servir el camp `Characteristics` per realitzar certes comprovacions en el nostre fitxer.

1.6.3 La capçalera "opcional"

Si bé aquest nom denota opcionalitat, això no és del tot cert en el cas d'executables i de DLL. És anomenada opcional perquè aquesta capçalera no és requerida en els fitxers de tipus objecte.

La capçalera opcional conté la major part de la informació important sobre la imatge de l'executable. Els camps més rellevants són els següents:

- `AdressOfEntryPoint:` La direcció del punt d'entrada relativa a la imatge base quan l'arxiu executable és carregat a memòria.
- `BaseOfCode:` *Offset* relatiu del codi (secció `.text` que descriurem més endavant) en la imatge carregada.
- `SizeOfCode:` Mida del codi executable.
- `SizeOfInitializedData:` Mida de les dades inicialitzades.
- `SizeOfUninitializedData:` Mida de les dades sense inicialitzar (secció `.bss`).
- `ImageBase:` L'adreça base preferida en l'espai de memòria del procés perquè la imatge de l'executable pugui ser assignada.
- `SizeOfImage:` La mida de la imatge carregada a memòria, incloent-hi totes les capçaleres.
- `SizeOfHeaders:` Aquest camp indica quant d'espai és utilitzat dins del fitxer per representar totes les capçaleres, incloent-hi la capçalera MS-DOS, la capçalera PE, la capçalera opcional i la capçalera de les seccions del PE.

La capçalera opcional, a més de tenir aquests camps, també conté directoris de dades, que indiquen on trobar components d'informació importants per a l'executable. Aquests directoris estan organitzats en *structs* de dues variables, de 8 bytes cada una, d'aquesta manera:

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

On `VirtualAddress` és l'adreça virtual relativa dins de la imatge on es troba la taula d'aquest directori, que és bàsicament el seu contingut, i `Size` és la mida d'aquesta taula. Dit d'una altra manera, la capçalera opcional indexa totes aquestes taules organitzant-les en directoris perquè puguin ser accedides més fàcilment.

De tots els directoris que existeixen, només ens interessa conèixer els tres següents:

- ***IMAGE_DIRECTORY_ENTRY_EXPORT***: Directori en el qual el seu camp `VirtualAddress` apunta a una taula anomenada **Export Table**. En aquesta taula és on apareixen totes les funcions exportades (normalment per arxius DLL) per ser utilitzades per altres mòduls.

- ***IMAGE_DIRECTORY_ENTRY_IMPORT***: Directori en el qual el seu camp `VirtualAddress` apunta a una taula anomenada **Import Address Table**, que és on apareixen totes les funcions importades de biblioteques, i adreces d'aquestes, juntament amb altres dependències externes, per l'executable.

- ***IMAGE_DIRECTORY_ENTRY_SECURITY***: Directori en el qual el seu camp `VirtualAddress` apunta a una taula (sense nom oficial) que conté els certificats digitals de l'executable.

1.6.4 Taula de seccions

Seguidament de la capçalera opcional, trobem la taula de seccions. Les seccions, bàsicament, són les que contenen el contingut de l'executable, com el codi, les dades, els recursos, etc. Cada una d'aquestes seccions té una capçalera i aquesta estarà dins la taula de seccions. Per tant, la taula de seccions està formada per un total de capçaleres equivalent al nombre total de seccions. Cada capçalera tindrà una mida de 40 bytes i el nombre total d'entrades dins d'aquesta taula que, tal i com s'ha mencionat prèviament, el podem trobar dins del camp `NumberOfSections` de la capçalera PE.

Dins de cada capçalera de secció podem trobar informació com: *a*) el camp `Name`, que indica el nom de la secció corresponent; *b*) el camp `VirtualAddress`, que indica l'adreça relativa de la imatge carregada a memòria en la que es carregarà la secció; *c*) el camp `VirtualSize`, que ens diu la mida de la secció un cop carregada a memòria; *d*) el camp `SizeOfRawData`, que ens dóna la mida de la secció en disc, i *e*) el camp `PointerToRawData`, que és un punter cap a l'inici de la secció.

1.6.5 Seccions

Normalment, una aplicació per a Windows NT té nou seccions predefinides anomenades: `.text, .bss, .rdata, .data, .rsrc, .edata, .idata, .pdata i .debug.` Algunes aplicacions no necessiten totes aquestes seccions, mentre que d'altres poden definir encara més seccions per satisfer les seves necessitats específiques. Els noms de les seccions poden ser modificats i se'n poden crear de noves, tot i que es recomana utilitzar els noms estàndard ja que modificar el nom de les seccions pot aixecar sospites als antivirus.

A continuació, es descriuen les seccions més rellevants per al nostre propòsit:

Secció de codi executable, `.text`:
Aquesta secció és la que conté el codi executable i és típicament de només lectura. El camp anomenat `BaseOfCode` és el que apunta a l'inici

d'aquesta secció i el camp `AdressOfEntryPoint` és el que apunta a la primera instrucció que s'executarà dins d'aquesta secció.

Seccions de dades, `.bss, .rdata, .data:`
La secció `.bss` representa totes les dades no inicialitzades incloent-hi les variables estàtiques.
La secció `.rdata` conté dades de només lectura, incloent-hi strings literals i constants. Per exemple, quan fem un `printf/cout`, l'*array* de caràcters que estem imprimint aniria en aquesta secció.
La secció `.data` conté tots els demés tipus de dades com, per exemple, les variables globals, variables estàtiques inicialitzades, etc.

Secció de recursos, `.rsrc:`
Aquesta secció conte informació sobre altres recursos necessaris per l'executable com, per exemple, la icona de l'arxiu o les fonts que utilitza. Aquesta secció és comunament utilitzada per afegir el codi del *malware* ja encriptat dins de l'*stub* que el desencriptarà i carregarà a memòria, tot i que hi ha d'altres alternatives. En el cas del *Proof of Concept* s'utilitzarà una altra alternativa que veurem més endavant.

2. Estat de l'art

En l'actualitat, més del 80% del *malware* existent està empaquetat amb un *crypter* i hi ha evidències que indiquen que el 50% dels nous virus són simplement versions reempaquetades dels ja existents (veure Figura 2.1) [5].

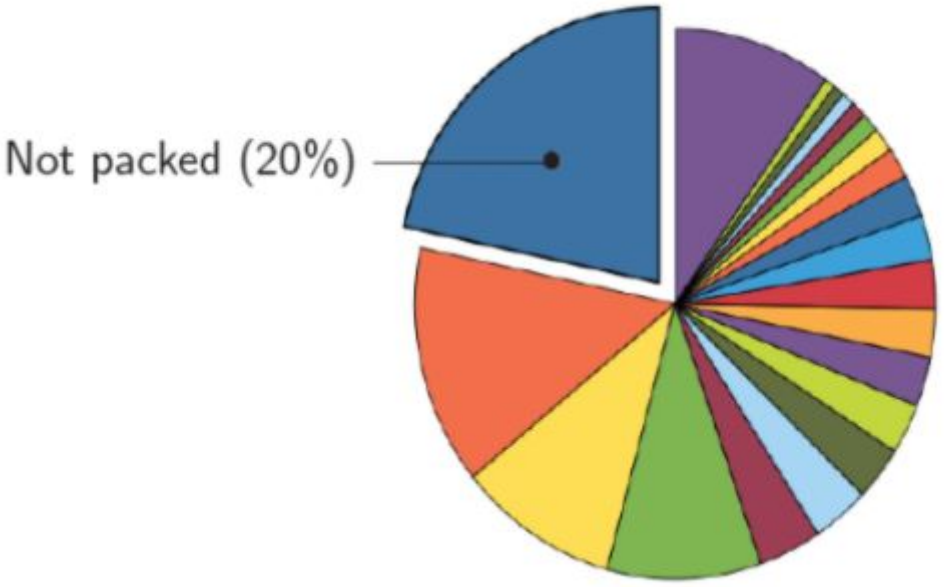

Figura 2.1 *"Malware and packing: 80% of new malware samples are packed with various packers, 50% of new malware samples are simply repacked versions of existing malware" [5]*

Dins del món del cibercrim, l'ús dels *Software Packers* és molt comú com acabo d'indicar, i la majoria de *malware* passa per un procés d'ofuscació abans de començar a ser distribuït. En molts fòrums *underground,* o comunitats *Black Hat*, és molt fàcil trobar un mercat dedicat exclusivament a la venda de serveis relacionats amb els *crypters*.

Dins d'aquestes comunitats podem trobar tant tarifes per a usos individuals, com a quotes per un nombre determinat de mesos o, inclús, de per vida. També és comú veure que s'ofereixen diferents serveis on, per exemple, el comprador envia el seu *malware* a l'ofertador i aquest li retorna l'executable ja encriptat i indetectable. Un altre servei és la venda

directa del *crypter* i, en aquest cas, el client mateix té accés directe al programa en si i pot reutilitzar-lo els cops que vulgui.

Cal dir també, que el desenvolupador ha d'anar actualitzant cada cert temps el seu *crypter* perquè sigui capaç de mantenir el seu nivell d'ofuscació a *FUD*, doncs els antivirus generen signatures i com que aquests *crypters* que es venen solen tenir molts clients, el temps de detecció sol ser molt menor. Els desenvolupadors de *crypters* són persones amb un gran coneixement del seu funcionament i es dediquen exclusivament a aquest tipus de mercats. En la Figura 2.2 en podem veure un exemple.

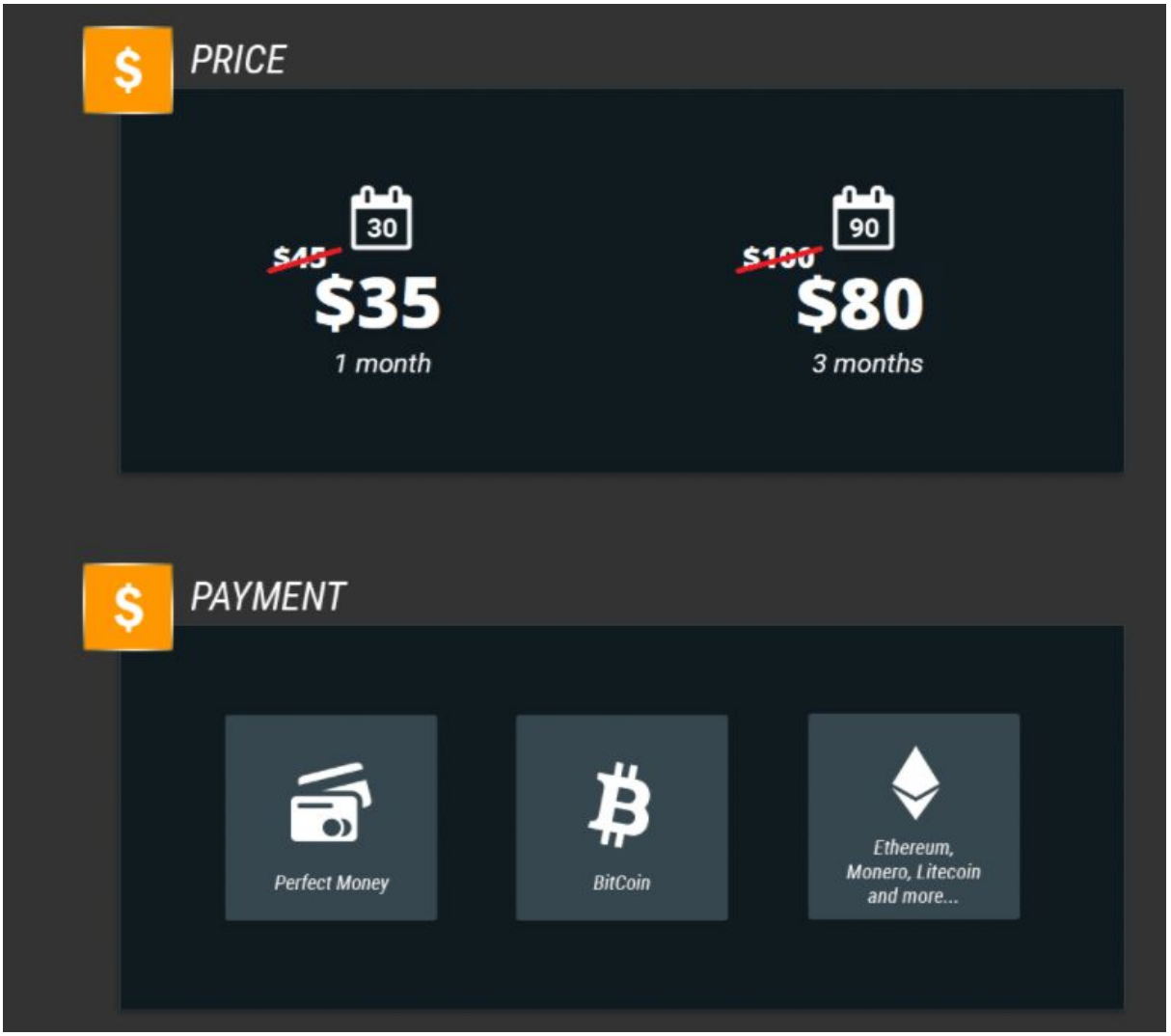

Figura 2.2 Publicació de l'usuari d'un fòrum *Black Hat* oferint la venda d'un *crypter* desenvolupat per ell o pel seu equip [6].

D'altra banda, les empreses d'antivirus estan constantment analitzant, mitjançant mètodes com l'enginyeria inversa, els nous *crypters* i *malwares* encriptats per aquests per veure les noves tècniques que van apareixent, per així buscar maneres de poder combatre'ls. De totes maneres, com he explicat anteriorment, el fet que un executable estigui ofuscat, fa que sigui molt difícil l'aplicar-li enginyeria inversa.

3. Formulació del problema

3.1 Problema

Si ens posicionem des del punt de vista d'un atacant, amb intencions de voler distribuir el seu *malware* (*malware spreading*), la seva preocupació inicial serà la d'aconseguir total impunitat davant la primera línia de defensa dels antivirus.

En els darrers anys, el mercat i l'ús dels antivirus està creixent molt ràpidament fins al punt que s'ha arribat a assolir un total de 23.000 milions de dòlars a tot el món l'any 2020 en la venda d'aquest tipus de software de ciberseguretat [7].

Tot i que inicialment pugui semblar que no sempre serà necessari l'ús d'eines d'ofuscació, ja que podem pensar que hi ha molts usuaris que no utilitzen antivirus, això no és del tot cert. Per exemple, el sistema operatiu de Windows ofereix el seu propi sistema de protecció anomenat Windows Defender que detectarà com a amenaça qualsevol arxiu maliciós del qual contingui informació a la seva base de dades. Per tant, si l'atacant pretén utilitzar un virus ja conegut per al seu propi benefici, l'única opció que té és passar-lo per un *crypter* (ja sigui acudint al mercat negre o bé implementat per ell mateix).

3.2 Objectius

L'objectiu principal d'aquest treball és el d'implementar un *crypter*, com a *Proof of Concept*, utilitzant algunes de les possibles tècniques, al mateix temps que anem adquirint coneixement del funcionament d'aquestes, i veure la seva eficàcia contra els antivirus moderns. Un subobjectiu d'aquest objectiu principal és intentar aconseguir la menor ràtio de detecció possible per qualsevol tipus d'arxiu maliciós. Un cop implementat, es buscaran altres maneres i alternatives per acabar de millorar l'ofuscació dels arxius de prova.

Dins dels objectius més específics entraria el desenvolupament d'una eina gràfica, ja sigui una *Graphical User Interface* (GUI) o una *WebApp*, i donar-li a l'usuari vàries opcions: escollir quin executable vol ofuscar, la clau

d’encriptació a utilitzar, escollir una icona per l’arxiu resultant, la signatura a copiar i el nom del fitxer de sortida.

Un cop implementat el *crypter* es farà una anàlisi comparativa utilitzant un servei web que actua com a *multiscanner* d’antivirus. Es farà servir un malware actualment conegut com a arxiu de prova i s’avaluaran les diferents tècniques d’ofuscació implementades.

El *multiscanner* [8] que s’utilitzarà consta d’un total de 26 antivirus. El meu objectiu inicial era el d’aconseguir reduir la ràtio de detecció de qualsevol *malware* actualment ja conegut i detectat pels antivirus a **3/26** - **5/26** (és a dir, d’entre 3 a 5 deteccions d’un total de 26 antivirus). Més endavant veurem que els resultats han estat molt més satisfactoris de l’inicialment esperat.

4. Gestió del projecte

4.1 Abast i possibles obstacles

L'abast d'aquest projecte comprèn des del proporcionar al lector una base teòrica dels conceptes necessaris i procediments més comuns dins de l'ofuscació de *malware*, fins a la implementació mateixa d'un *crypter* així com l'explicació de la presa de decisions que s'han fet durant el seu desenvolupament. Per tant, doncs, la intenció serà donar una visió completa de tot el procés i intentar aconseguir els millors resultats possibles.

Tot i que realment es podria fer en ambdues arquitectures, el desenvolupament serà sota l'arquitectura de x86 i no x64, ja que per demostrar el concepte no cal centrar-nos en temes de compatibilitat. La diferència entre les dues arquitectures, per la nostra implementació, només afectaria a alguns dels camps dins del format PE.

El que no abastarà aquest projecte seran certs extres dins el *crypter* que s'ofereixen en els mercats negres amb motivació comercial com, per exemple: *a*) el polimorfisme en codi (perquè així hi hagi una única generació de l'*stub* per a cada client); *b*) una GUI avançada; *c*) copiar informació del *assembly* d'un executable (versió, nom de l'empresa, informació, etc.), o *d*) que s'iniciï l'executable automàticament cada cop a l'engegar l'ordinador (normalment, la majoria de *malwares* ja porten aquesta opció per defecte però els *crypters* comercials ho ofereixen per si de cas).

Els possibles obstacles que poden sorgir són a l'hora d'implementar el mètode d'injecció dinàmica a memòria de l'executable ja desencriptat, doncs requerirà entendre molt bé el funcionament del format del PE, ja que serà necessari saber com recórrer les seves estructures internes. Un altre possible problema serà el d'intentar millorar les ràtios de detecció obtingudes un cop aconseguim un *crypter* funcional, si aquestes no han estat satisfactòries, ja que això significarà que serà necessari investigar quins mètodes d'ofuscació addicionals podem incorporar al nostre programa.

4.2 Riscs

De cara a les proves finals, quan s'utilitzi *malware* real, això pot suposar un risc per al meu ordinador. Per tant, serà necessari l'ús de màquines virtuals com a mètode de prevenció. S'ha de tenir en ment que algunes de les proves consistiran en executar *malware* real que hagi passat pel meu *crypter* per validar el seu funcionament i extreure'n conclusions.

Un altre risc amb el que em puc trobar és el d'utilitzar un *multiscanner* que distribueixi els resultats a les empreses d'antivirus. Hi ha *multiscanners* que ofereixen l'opció de no distribuir i això, per al nostre desenvolupament, és preferible ja que ens mostren els resultats de l'escaneig i res més. El fet que un arxiu (sigui l'arxiu que sigui, *malware* o no) que hagi passat pel nostre *crypter* (i, per tant, contindrà el nostre *stub*), sigui escanejat en un *multiscanner* que distribueix, pot causar un impacte negatiu al nostre projecte, ja que per si qualsevol cosa alguna d'aquestes empreses decideix fer una anàlisi manual (o utilitzant algunes altres eines, les quals desconec) del nostre arxiu i detecta l'ús d'un *crypter*, això podria significar la generació d'una signatura a la seva base de dades del nostre programa, de tal manera que ens veuríem constantment obligats a modificar el plantejament del nostre codi intern i fer molt de *refactoring*. Per tant, serà important trobar un *multiscanner* que ens asseguri que no distribueix els seus resultats.

Finalment, aquest és un projecte amb dependències externes, sent la principal l'API de Windows. Això significa que aquest projecte requerirà d'un manteniment, ja que si per qualsevol motiu algunes funcions de l'API o paràmetres d'aquestes o *structs* interns varien, caldrà fer les modificacions corresponents. És per aquest motiu també que les persones que es dediquen a aquests negocis ofereixen aquest tipus de suport a les seves tarifes, de tal manera que cap dels seus potencials clients pugui quedar-se sense el servei.

4.3 Metodologia de treball

Per realitzar aquest projecte s'ha decidit, com a primer pas, dur a terme un estudi preliminar del funcionament intern dels *crypters* mitjançant articles acadèmics i fer una anàlisi d'alguns dels pocs codis font públics que hi ha disponibles, amb l'objectiu d'entendre els conceptes més rellevants.

Atès que aquest projecte no consisteix en construir un servei per a un suposat client, sinó en la implementació d'un concepte en si, i els requisits són autoimposats a mesura que es va avançant en el projecte, la fase de requeriments ha estat construïda dinàmicament al mateix temps que s'ha anat implementant el projecte i que s'han anat adquirint nous coneixements. Per tant, la metodologia que s'ha dut a terme és la que correspon al model de **Desenvolupament iteratiu i incremental** [9]. En la Figura 4.1 es pot trobar una representació gràfica d'aquest model. Utilitzar aquest tipus de model m'ha permès reforçar tots els conceptes que anava aprenent al mateix temps que anava avançant cap a l'objectiu final. El fet de complementar l'estudi teòric amb un desenvolupament iteratiu m'ha ajudat a consolidar tots els conceptes i anar avançant d'una manera més dinàmica.

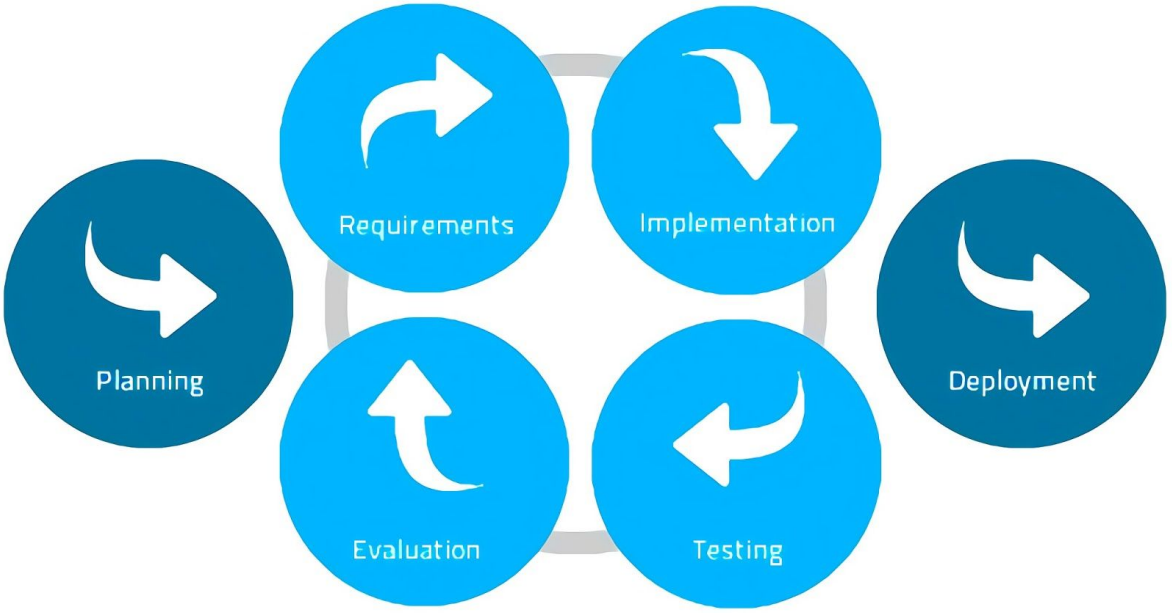

Figura 4.1 Model de Desenvolupament iteratiu i incremental que ha estat seguit durant el projecte. [10]

A mode d'exemple, per motius de claredat, **una iteració** d'aquest model dins del projecte podria ser la següent:

1. **Requeriment:** S'ha d'implementar un algoritme d'encriptació en RC4 per utilitzar-ho en els nostres projectes.
2. **Implementació:** Programació d'aquest algoritme en C++ utilitzant el IDE de Visual Studio 2019.
3. **Testing:** Proves d'encriptació utilitzant strings i comprovant els resultats obtinguts amb els esperats utilitzant eines online com *https://www.dcode.fr/rc4-cipher*.
4. **Avaluació:** Comprovar si els resultats són satisfactoris abans de decidir els següents requeriments. Potser l'algoritme no ens convenç i volem provar-ne un altre, potser no ens en sortim, o potser va tot bé i seguim endavant en el projecte.

A cada iteració, serà la fase d'avaluació la que determinarà l'estat futur del projecte.

En l'etapa inicial del projecte, la planificació (*Planning*) contempla tota la preparació necessària, des de l'estudi de la part teòrica i conceptual fins a la preparació d'un *roadmap* distribuït en subtasques genèriques. En l'etapa final del projecte, el desplegament (*Deployment*) bàsicament consistirà en unificar el *crypter* implementat amb la *webapp* de manera funcional.

4.4 Planificació temporal

A partir de la metodologia de treball que s'ha seguit en aquest projecte, la planificació temporal i les tasques han estat distribuïdes de tal manera que permetin avançar d'una manera progressiva i incremental.

El treball s'ha dividit en quatre fases principals:

1. Fase inicial: En aquesta fase s'ha buscat tota la informació necessària i s'ha estudiat part de la teoria essencial per iniciar-se en el projecte.
2. Fase d'anàlisi i disseny: En aquesta fase s'ha definit què és el que es farà i fins on s'arribarà. També, s'han escollit les tecnologies que s'utilitzaran, s'ha seguit llegint i estudiant bibliografia per veure què és i què no és possible fer i s'han marcat els objectius.

3. Fase de desenvolupament: En aquesta fase s'han implementat tots els projectes que conformen el *crypter* i la *webapp* gràcies al coneixement obtingut de les fases anteriors. També s'ha seguit buscant informació i estudiant, per dur a terme certes parts clau del *crypter*.
4. Fase de tests: En aquesta fase s'han anat realitzant tots els tests necessaris pel desenvolupament per tal d'assegurar-nos el correcte funcionament. També s'han realitzat els tests finals, introduint diversos canvis, amb l'objectiu d'analitzar els resultats i treure conclusions per al treball.
5. Fase de documentació: Tota la documentació s'ha anat redactant durant aquesta fase.

Com es pot veure a la Taula 4.1, la distribució temporal de les tasques entre les fases ha estat bastant intercalada entre moltes d'elles degut a la metodologia de treball seguida descrita en l'apartat anterior.

Tasca	**Inici**	**Fi**	**Duració en hores**
Fase Inicial	**01/09/20**	**21/10/20**	**50 hores**
Cerca d'informació	01/09/20	24/09/20	18 hores
Estudi teòric de conceptes	24/09/20	14/10/20	26 hores
Cerca i lectura de crypters a GitHub	08/10/20	21/10/20	6 hores
Fase d'analisi i disseny	**19/10/20**	**16/12/20**	**40 hores**
Disseny del *builder*	19/10/20	21/10/20	6 hores
Disseny de l'*stub*	26/10/20	05/11/20	22 hores
Disseny del *SignatureClone*	20/11/20	23/11/20	8 hores
Disseny de la *webapp*	15/12/20	16/12/20	4 hores
Fase de desenvolupament	**22/10/20**	**07/01/21**	**106 hores**
Implementació del *builder*	22/10/20	26/10/20	12 hores
Implementació de l'*stub*	06/11/20	07/01/21	60 hores
Implementació del SignatureClone	23/11/20	25/11/20	10 hores
Implementació del *frontend*	17/12/20	25/12/20	14 hores
Implementació del *backend*	22/12/20	25/12/20	10 hores
Fase de tests	**20/10/20**	**20/01/21**	**24 hores**

Tests de desenvolupament	20/10/20	07/01/21	18 hores
Tests finals	14/01/21	20/01/21	6 hores
Fase de documentació	**06/10/20**	**27/01/21**	**70 hores**
Total	**01/09/20**	**27/01/21**	**290 hores**

Taula 4.1 Distribució de tasques i dedicació d'hores per a cadascuna.

En la Figura 4.2 es pot apreciar una representació visual amb un diagrama de Gantt del *workflow* entre les fases i tasques d'aquestes. A l'annex s'hi pot trobar aquest mateix diagrama de Gantt ampliat.

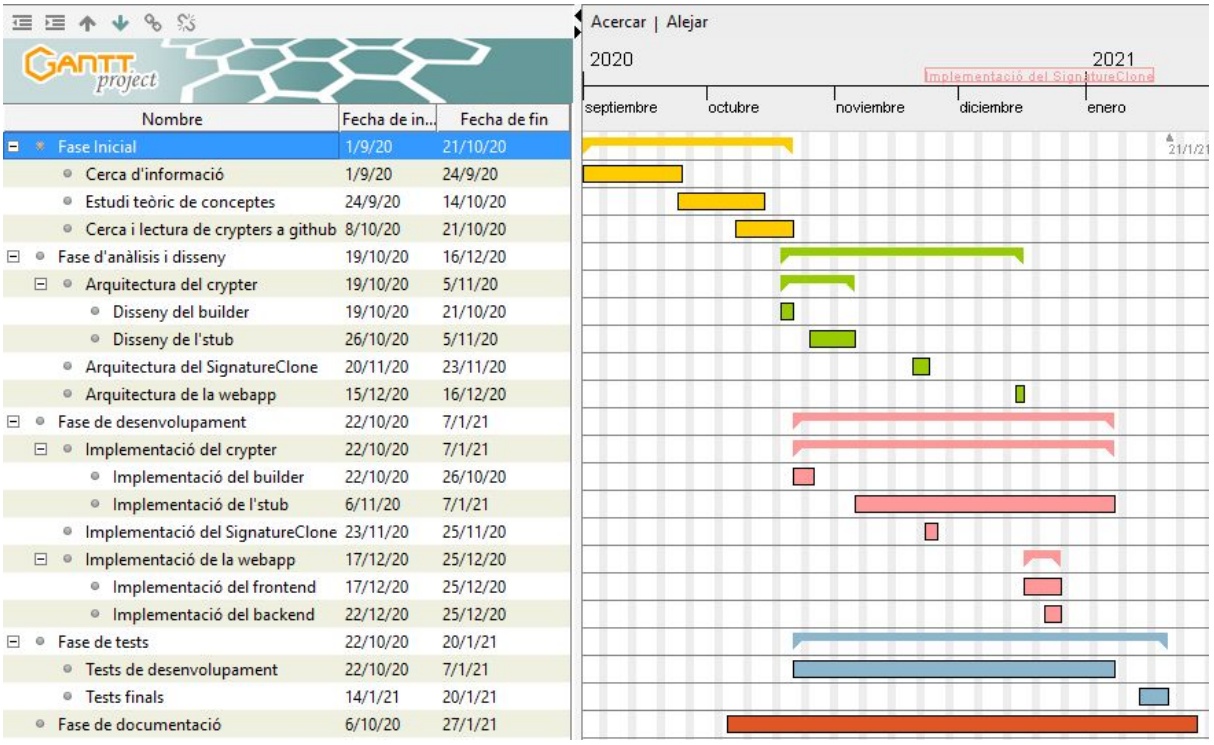

Figura 4.2 Diagrama de Gantt.

4.5 Recursos

Els recursos utilitzats per dur a terme aquest projecte es troben dividits en els següents apartats: recursos humans, recursos de hardware i recursos de software.

4.5.1 Recursos humans

Els diversos rols que es podrien considerar per aquest projecte són els que es mostren en la Taula 4.2.

Rol	**Descripció**
Cap de projecte	Encarregat de decidir quines tasques es realitzaran, de quina manera i amb quina prioritat. També serà el responsable de la documentació de tot el projecte.
Desenvolupador	Responsable de la implementació de *crypter* i de la *webapp* que li donarà suport com a GUI.
Tester	Encarregat de realitzar tot tipus de proves per assegurar un correcte funcionament del programa i fer tests dels resultats obtinguts per contrastar-los amb els esperats.

Taula 4.2 Rols i descripció dels recursos humans d'aquest projecte.

En aquest cas, en ser un projecte individual, he portat jo mateix a terme el paper de tots els rols i, per tant, he dut a terme totes les tasques assignades.

4.5.2 Recursos de hardware

La Taula 4.3 mostra els recursos materials utilitzats per poder dur a terme el projecte.

Recurs	Finalitat
Ordinador amb *S.O.* Windows	Desenvolupament del projecte, realitzar les proves necessàries, redactar, etc.
Router	Proporcionar accés a Internet per la cerca d'informació necessària per al projecte i la realització de proves amb els *multiscanners*.

Taula 4.3 Recursos hardware emprats en aquests projecte i la seva finalitat.

4.5.3 Recursos de software

La Taula 4.4 mostra el software utilitzat tant pel desenvolupament com per la documentació del projecte.

Recurs	Finalitat
Visual Studio 2019	IDE utilitzat per la implementació del *crypter* del projecte.
Visual Studio Code	Editor de text utilitzat per la implementació de la *webapp* del projecte.
Postman	Eina utilitzada per la comprovació del correcte funcionament dels *endpoints* de l'API de la *webapp*.
Flask	*Framework* de Python utilitzat per crear el *backend* de la *webapp* del projecte.
ReactJS	*Framework* de JavaScript utilitzat per al desenvolupament del *frontend* de la *webapp* del projecte.
Github	Eina utilitzada pel manteniment del codi del projecte.
Google Drive	Eina utilitzada per redactar la documentació de tot el projecte.

Gantt	Eina utilitzada per representar gràficament la distribució temporal de les tasques del projecte.
AntiScan.Me	Pàgina web utilitzada com a *multiscanner*.

Taula 4.4 Recursos software emprats en aquests projecte i la seva finalitat.

4.6 Gestió econòmica

Els costos d'aquest projecte deriven directament dels recursos utilitzats.

Respecte als rols emprats dins dels recursos humans, segons les estadístiques mitjanes dels sous pels llocs de treball al voltant de l'àrea de Barcelona, l'any 2020, trobem les dades mostrades en la Taula 4.5.

Rol	**Sou brut anual mitjà**	**Sou brut mensual mitjà**	**Sou brut hora**
Cap de projecte	47.000€	4.000€	25€
Desenvolupador	30.000€	2.500€	15,6€
Tester	28.000€	2.300€	14,4€

Taula 4.5 Sous bruts mensuals, en mitjana, i càlcul estimat de sou per hora per a una jornada habitual de 40 hores setmanals pels diferents rols participants en el projecte. Dades extretes de [11].

La Taula 4.6 mostra el cost dels recursos humans obtinguts a partir de la planificació temporal que s'ha dut a terme en aquest projecte segons les hores dedicades per rol i tenint en compte les dades de la taula anterior.

Rol	**Hores dedicades**	**Preu total**
Cap de projecte	160 hores	4.000€
Desenvolupador	106 hores	1653€
Tester	24 hores	345€
Total	**290 hores**	**5.998€**

Taula 4.6 Cost dels recursos humans emprats en el projecte.

La Taula 4.7 mostra els costos dels recursos de hardware que s'han utilitzat en el desenvolupament d'aquest projecte.

Recurs	**Cost**
Ordinador	1.100€
Router amb accés a internet	45€ mensuals

Taula 4.7 Cost dels recursos hardware emprats en el projecte.

Tenint en compte que la durada del projecte ha estat de quatre mesos, l'import total del recursos de hardware és de **1280€**.

Finalment, dins dels recursos de software, els únics possibles costos detectats són els d'alguns *multiscanners* a l'hora de realitzar proves, ja que la resta de programari utilitzat és d'ús lliure i gratuït. Hi ha diversos serveis d'anàlisi d'antivirus online i diferents tarifes segons les diferents opcions possibles a escollir. En la Figura 4.3 es pot apreciar un exemple de diferents tarifes d'un d'aquests multicanners.

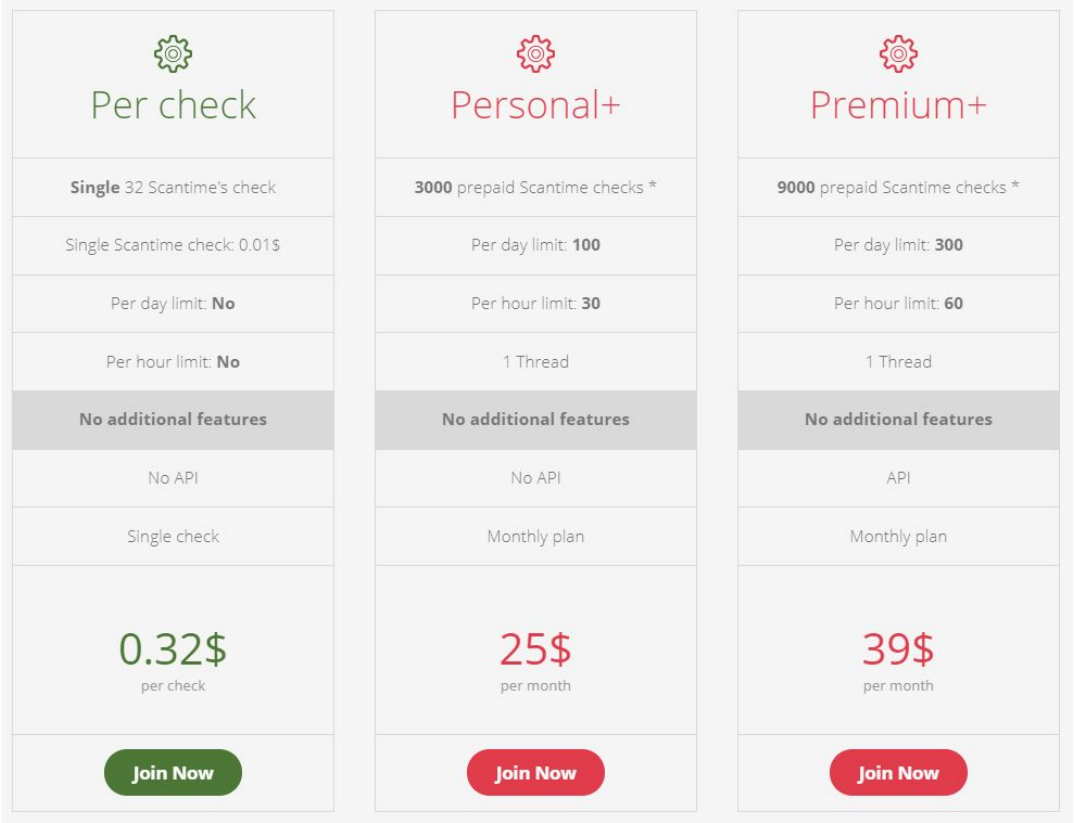

Figura 4.3 Tarifes de diferents plans d'una pàgina web que ofereix el servei de *multiscanner*.

De totes maneres, finalment, el cost d'ús d'aquests *multiscanners* no serà necessari ja que algunes pàgines (com la que utilitzarem) ofereixen fins a tres *multiscans* gratuïts i, utilitzant eines que proporcionen accés VPN (*Virtual Private Networks*), podem arribar a tenir escanejos gratuïts de manera indefinida. Per tant, el cost dels recursos software sumarà un total de **0€**.

El cost total del projecte acabarà sent d'un total de 5.998€ + 1280€ = **7.278€**.

5. Especificació

Com ja s'ha mencionat anteriorment, per la metodologia emprada en aquest projecte, alguns requeriments han anat apareixent a mesura que s'anava avançant en aquest. De totes maneres, podem definir un conjunt de requeriments generals, que són els següents:

- Per construir un *crypter* serà necessària la implementació de dos projectes separats, on la sortida de l'execució del primer serà part de l'entrada de l'execució del segon. El primer projecte serà el ***Builder*** i el segon projecte serà la construcció de l'***Stub***.

- El ***Builder*** ha de ser capaç de llegir un arxiu binari i, utilitzant qualsevol algoritme d'encriptació, encriptar tots els bytes d'aquest i fer un *output* de la sortida en un fitxer capçalera amb extensió **`.h`**.

- L'***Stub*** haurà d'agafar aquest fitxer **`.h`** de sortida de la capçalera i desencriptar el seu contingut, per carregar aquest arxiu a memòria dinàmica utilitzant el mètode conegut com a *Dynamic Forking* o *Process Hollowing*.

- La compatibilitat d'aquest projecte serà exclusivament per a l'arquitectura x86.

- A més del *crypter*, per millorar els nostres resultats finals de les ràtios de detecció, s'haurà d'implementar un tercer projecte anomenat ***SignatureClone*** on l'objectiu principal serà el de clonar el certificat digital d'un executable reconegut al nostre (veure Figura 5.1).

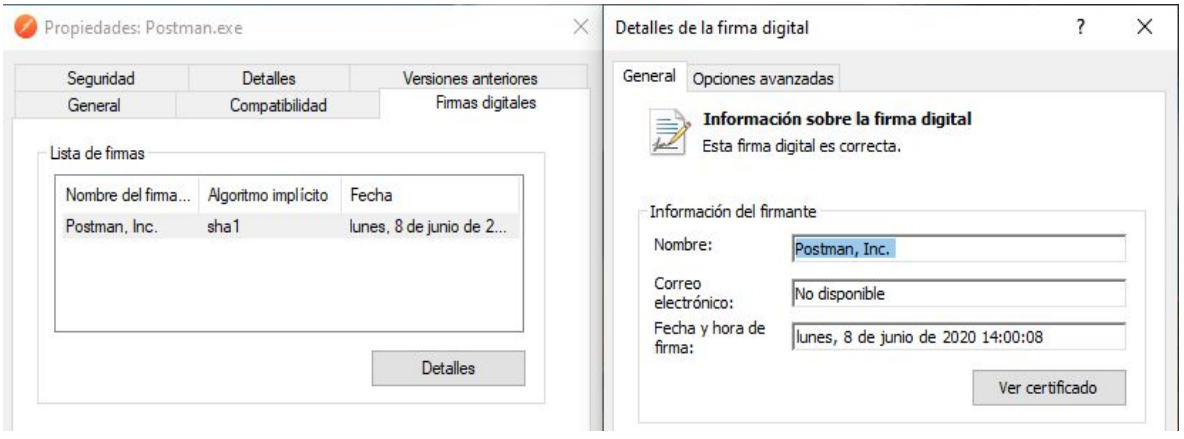

Figura 5.1 L'opció "Propietats" d'un arxiu executable que conté una o més firmes digitals, permet trobar-les en seleccionar la pestanya anomenada "Firmes digitals". Aquest certificat serà el que serà clonat al nostre *stub* resultat del *crypter*.

- És necessària la implementació de tècniques d'ofuscació addicionals al *crypter* per a millorar els resultats d'aquest.

- S'espera poder reduir mínim un 80% el nombre de deteccions dels antivirus sobre el total de deteccions actual per a qualsevol malware.

- La *webapp* ha d'atorgar a l'usuari la capacitat de penjar un arxiu executable x86 perquè passi pel *crypter*. També se li donarà vàries opcions a escollir, com el nom de l'executable final, la icona de l'executable resultant, la clau d'encriptació a utilitzar (amb possibilitat d'autogeneració), l'executable del que es vol clonar el certificat i el nom del fitxer de sortida.

6. Disseny

En aquest apartat s'explicaran amb detall totes les decisions preses en la implementació d'aquest projecte, el funcionament d'aquestes i les tecnologies usades.

6.1 Arquitectura del *crypter*

Com ja s'ha explicat en apartats anteriors, un *crypter* consta de dues parts principals, el *Builder* i l'*Stub*, sent aquesta última la més important.

6.1.1 *Builder*

Recordem que la funció principal del *builder* és la d'encriptar els bytes de l'arxiu executable que rebi com entrada el programa, perquè aquests siguin utilitzats posteriorment en l'*stub*.

Per construir-lo hi ha vàries possibles opcions i haurem de decidir entre totes aquestes.

Primerament, haurem d'escollir quin és l'algoritme d'encriptació que volem utilitzar. Inicialment, jo volia donar-li a l'usuari la possibilitat d'escollir, en la *webapp,* quin algoritme utilitzar a l'hora d'encriptar el seu executable (per tant, implementar vàries opcions d'algoritmes), però després de fer diverses proves, em vaig adonar que l'algoritme que s'utilitzi és irrellevant mentre els bytes maliciosos siguin encriptats. Això és perquè els antivirus no perden temps en intentar desencriptar amb força bruta els arxius (tot i que detectin que possiblement estan encriptats), ja que han de ser ràpids i eficients, i perdre massa temps en un sol arxiu no és adequat. Tenint en compte això, doncs, inclús l'algoritme d'encriptació més senzill, per exemple el *XOR*, ja ens serviria. En el cas del *crypter* d'aquest projecte s'ha decidit utilitzar l'algoritme d'encriptació anomenat ***RC4*** [12].

L'altra decisió que s'ha de prendre és la de com tractar els bytes de l'executable un cop encriptats perquè l'*stub* pugui accedir a ells. Hi ha dues possibilitats, cadascuna amb els seus avantatges i inconvenients:

1. La primera opció és la de crear un nou arxiu resultant amb extensió `.h` (capçalera de `C++`), que contingui un *array* de caràcters, perquè després el projecte de l'*stub* l'utilitzi i el pugui desencriptar. L'avantatge d'aquest mètode és que no estem utilitzant cap crida addicional a l'API de Windows per tractar la variable que conté els bytes encriptats. L'inconvenient és que cada cop que es vulgui encriptar un nou executable, l'*stub* haurà de ser recompilat amb el nou arxiu capçalera `.h` generat pel *builder* (ja que contindrà una variable *array* de caràcters amb contingut diferent).

2. L'altra opció és la d'afegir aquestos bytes com a recurs (com seria una simple icona, però en aquest cas un *array* de caràcters) al segon projecte, el de l'*stub*, de tal manera que puguin ser utilitzats de manera fàcil amb una crida a l'API de Windows anomenada ***LoadResource*** [13]. L'avantatge d'utilitzar aquesta tècnica és que l'*stub* no cal que sigui recompilat cada cop que es vulgui encriptar un executable diferent, ja que sempre serà el mateix (a no ser que s'utilitzi poliformisme, que llavors sí que caldria generar cada *stub* amb una nova compilació) perquè l'únic que està sent modificat és un recurs extern, no cap variable de dins del codi. L'inconvenient d'aquesta tècnica és que s'està utilitzant una crida addicional a l'API de Windows que probablement entri dins de molts patrons heurístics d'implementacions de *crypters*. Des del meu punt de vista, l'ideal és intentar utilitzar el menor nombre de crides a l'API de Windows que es puguin relacionar amb algunes de les implementacions ja conegudes pels antivirus dels *crypters* existents.

Tenint en compte aquestes dues opcions, s'ha decidit sacrificar la comoditat que proporciona la segona opció (no haver de compilar cada cop l'*stub*), a canvi d'estalviar-nos utilitzar una crida a l'API de Windows addicional. Per tant, s'ha utilitzat la primera opció.

Addicionalment al procés d'encriptació, a causa de l'elevada entropia resultant d'encriptar els bytes (que no deixen de ser caràcters unicode molt poc comuns formant seqüències amb molta aleatorietat), s'ha decidit afegir una capa addicional amb una codificació dels bytes encriptats en **Base64** [14]. Això és molt útil ja que els bytes resultants consistiran únicament en una representació dels caràcters A-Z, a-z i 0-9 (un total de 62 dígits, amb els dos últims, que sumaran un total de 64, sent variables segons el tipus de variant de Base64 utilitzat). Utilitzar aquesta codificació redueix moltíssim l'entropia i això és quelcom positiu, ja que quan un

antivirus detecta una elevada entropía en la informació binària d'un executable, pot marcar-lo com a sospitós.

La representació gràfica del *builder* de la Figura 6.1 pot ajudar a entendre millor el seu funcionament.

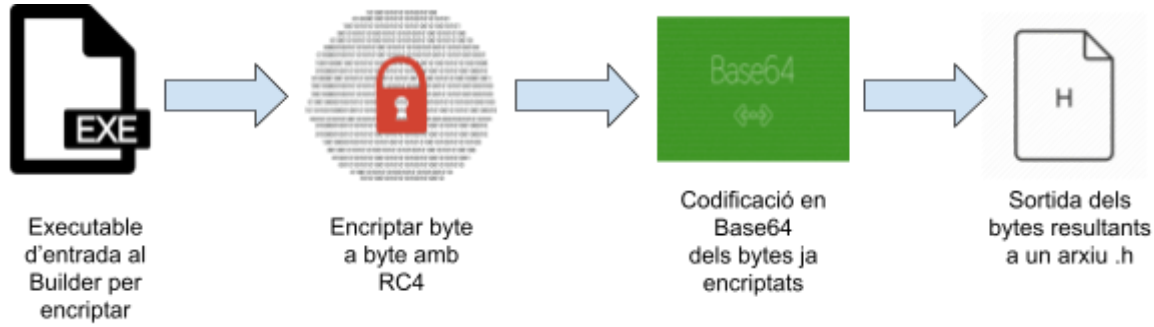

Figura 6.1 Representació gràfica del projecte *Builder* del *crypter*.

6.1.2 *Stub*

L'*stub* és la part més important del *crypter*, ja que serà l'encarregat de desencriptar els bytes de l'arxiu executable encriptat prèviament pel *Builder* i carregar la imatge dinàmicament a memòria sense tocar el disc dur. Per poder fer això, existeixen diversos mètodes diferents del que s'anomena **Injecció dinàmica**. En aquest apartat, s'explicarà el mètode d'injecció utilitzat en aquest treball, anomenat *Process Hollowing* o *Dynamic Forking*, i a l'annex s'explicaran altres alternatives i mètodes a utilitzar per al mateix propòsit. També s'explicaran tècniques addicionals implementades amb l'objectiu de reduir les ràtios de detecció.

Primerament, l'*stub* ha d'accedir a la variable que conté l'*array* de caràcters (bytes encriptats) de l'executable que ha passat prèviament pel *builder,* descodificar en **Base64** aquests bytes i, després, utilitzar el mateix algoritme d'encriptació (**RC4** en aquest cas) i la mateixa clau, per desencriptar els bytes descodificats.

6.1.2.1 *Process Hollowing*

Un cop ja tenim els bytes de l'arxiu binari original, hem de procedir a carregar-los dinàmicament a memòria sense tocar el disc dur. Això ho aconseguirem amb la tècnica anomenada *Process Hollowing* que consta, de manera general, dels següents passos:

1. Crear un nou procés en un estat suspès. En aquest cas, el procés que es crearà és una còpia del mateix procés.
2. Modificar-ne el contingut amb la imatge d'un altre executable (en aquest cas, pels bytes ja desencriptats).
3. Fer les modificacions necessàries perquè l'execució de la imatge en el nou procés sigui correcta.
4. Reprendre l'execució del procés suspès.

La representació gràfica d'aquest mètode de la Figura 6.2 pot ajudar a entendre millor el seu funcionament.

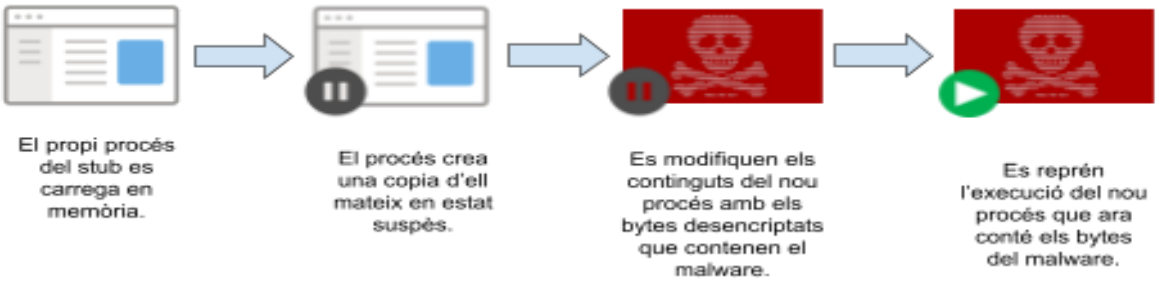

Figura 6.2 Representació gràfica del mètode d'injecció en memòria ***Process Hollowing***.

A continuació, es descriuen de manera més detallada els passos que s'han seguit per obtenir el correcte funcionament de l'*stub*. Per entendre millor els procediments duts a terme, és molt recomanable haver llegit en detall i entès l'apartat **1.6** que tracta del format PE (***Portable Executable***).

1. Guardar les capçaleres del PE del *malware* desencriptat

A l'apartat 1.6 s'ha descrit el format dels PE, que ens permetrà recórrer tota l'estructura interna d'un fitxer executable de Windows i fer les modificacions que calguin.

L'objectiu del primer pas és el d'obtenir accés a les capçaleres del *malware* (bytes desencriptats) mitjançant els tipus natius de l'API de Windows (***IMAGE_DOS_HEADER*** i ***IMAGE_NT_HEADERS*** [15]). Recordem que la capçalera DOS és la primera que trobem dins d'un executable i, per accedir a la capçalera NT (també anomenada capçalera PE), s'ha d'utilitzar el camp anomenat ***e_lfanew*** dins de l'*struct* de la capçalera DOS, tal i com es descriu a l'apartat **1.6.2**. Per tant, primer accedim a la capçalera ***IMAGE_DOS_HEADER*** i després, utilitzant el camp anomenat ***e_lfanew*** d'aquesta, podrem accedir a la segona capçalera ***IMAGE_NT_HEADERS***. Aquesta part és essencial ja que un cop tenim les dues capçaleres emmagatzemades (en variables, en aquest cas), podrem accedir a tots els camps necessaris per dur a terme la tècnica del *Process Hollowing* i fer les modificacions futures necessàries.

2. Crear un nou procés i obtenir el seu context

Un cop tenim ja les capçaleres, podem procedir a crear un nou procés, que en aquest cas serà una còpia del mateix procés, en estat suspès al que bolcarem tots els bytes del *malware*. Per fer això, utilitzarem la crida a l'API anomenada ***CreateProcessA*** [16] amb el flag *CREATE_SUSPENDED*.

Seguidament, serà necessari emmagatzemar el context del nou procés en una variable, ja que caldrà en el futur per establir el nou punt d'entrada abans de reprendre l'execució d'aquest procés suspès. Per poder accedir al context utilitzem la crida a l'API ***GetThreadContext*** [17]. El context del procés és un *struct* que conté informació de l'estat actual del procés mateix, així com de l'espai d'adreces, l'*stack*, l'estat dels registres, etc. En el nostre cas només ens interessen dos camps: *a*) el del registre `EAX` dins de l'*struct* del context, què és el que indica el punt d'entrada del programa (el modificarem més endavant), i *b*) el del registre `EBX`, un camp al que si li sumem 8 bytes (EBX + 8) [18], ens permet accedir a la direcció de la imatge base del procés. Aquí cal diferenciar entre la direcció de l'imatge base i la direcció del punt d'entrada: la primera, és la direcció del primer byte de l'executable a memòria, i la segona és la direcció relativa a aquesta, que indica la primera instrucció que s'executarà d'aquest programa.

3. Desmapejar el procés i establir la nova direcció base de la imatge

El següent pas consisteix en agafar el procés nou que hem creat i "desmapejar" *(unmap)* el seu contingut per poder reservar l'espai necessari pels bytes desencriptats que contenen el *malware*. Per fer això, farem ús de les crides ***NtUnmapViewOfSection*** [19] seguida de ***VirtualAllocEx*** [20].

Bàsicament, quan hem creat el nou procés al pas anterior, aquest conté a memòria la mateixa informació del procés que l'ha creat, ja que n'és una còpia. Per tant, hem d'alliberar aquest espai de memòria reservat per poder tornar a reservar-lo pels bytes que de veritat ens interessen: els del *payload* del *malware*. La mida d'aquests bytes la podem obtenir del camp ***SizeOfImage*** de la capçalera opcional obtinguda en el primer pas. Un cop reservem aquests bytes, també haurem de guardar (en una variable anomenada ***imageBase*** en aquest cas) el valor resultant de la funció ***VirtualAllocEx***, ja que retorna la direcció base del nou espai reservat.

4. Escriure els bytes del *malware* al nou procés

Ara que ja tenim el nou procés creat amb l'espai reservat suficient pels bytes del *malware*, podem procedir a escriure el seu contingut. Això ho fem amb la crida a l'API anomenada ***WriteProcessMemory*** [21]. Primerament, haurem de copiar les capçaleres dels bytes del *malware* al nou procés. Això ho farem agafant la variable ***imageBase*** com a punt d'escriptura inicial obtinguda al pas anterior, i utilitzant com a mida el camp ***SizeOfHeaders*** obtingut de la capçalera opcional. Com podem veure, el primer pas, el d'obtenir les capçaleres, és essencial, ja que ens està proporcionant molts camps útils a l'hora de manipular el nou procés amb els nostres bytes del *malware*.

Un cop ja hem escrit les capçaleres dels bytes del *malware* al nou procés, hem d'escriure les seves seccions, que són les que emmagatzemen el contingut de l'executable en si (codi, recursos, dades, etc.). Per poder escriure aquestes seccions haurem d'iterar tantes seccions com tingui el nostre *malware* (això dependrà del tipus de *malware* o programa amb el que estem tractant). El nombre de seccions ve donat, un altre cop, pel camp anomenat ***NumberOfSections*** de les capçaleres del PE. En aquest cas, el punt d'escriptura inicial no serà el de la variable ***imageBase***, ja que ja hem escrit al procés totes les capçaleres i les seccions s'han d'escriure a continuació. Segons l'estructura del format PE, sabem que les seccions estan dividides en capçaleres i contingut. Hi ha un total de capçaleres de seccions igual al ***NumberOfSections*** (és a dir, cada secció té la seva propia capçalera), i cada capçalera de secció ocupa 40 bytes.

Per tant, cada capçalera de secció corresponent la podrem trobar a:

```
for(i = 0; i<NumberOfSections;i++){
sectionHeader= sizeof(PEHeaders) + i * sizeof(IMAGE_SECTION_HEADER)
}
```

Aquestes capçaleres de seccions, però, no les escriurem com a tals al nostre nou procés sinó que, gràcies a la informació que contenen els seus camps, podrem escriure el contingut de les seccions directament. Els camps rellevants de cada ***sectionHeader*** són el ***VirtualAddress*** (que indica l'adreça relativa de la imatge carregada a memòria en la que es carregarà la secció), ***PointerToRawData*** (que és un punter cap a l'inici de la secció) i ***SizeOfRawData*** (que ens dona la mida de la secció a disc). Per tant, sabem que per a cada secció del nostre nou procés, haurem d'escriure a ***imageBase + VirtualAddress*** el contingut al que apunta ***PointerToRawData*** un total de ***SizeOfRawData*** bytes. El resultat final d'aquesta part, en pseudocodi, queda així:

```
for(i = 0; i<NumberOfSections;i++){
 secHeader= sizeof(PEHeaders) + i * sizeof(IMAGE_SECTION_HEADER);

 WriteProcessMemory(newProcess,imageBase+secHeader->VirtualAddress,
decryptedData+secHeader->PointerToRawData),secHeader->SizeOfRawData,0);
}
```

On **newProcess** serà el nou procés al que estem injectant el *malware*, **imageBase+secHeader->VirtualAddress** serà l'adreça on escriurem la secció d'aquesta iteració (dins d'aquest nou procés), **decryptedData+secHeader->PointerToRawData** serà el punter que apunta a l'inici del contingut de la secció que escriurem i **secHeader->SizeOfRawData** serà la mida d'aquesta secció actual en bytes.

Un cop fem això per totes les seccions, ja haurem injectat tots els bytes del *malware* al nostre nou procés.

5. Aplicar les relocalitzacions, en cas que sigui necessari

Un executable en Windows pot estar compilat utilitzant el que s'anomena en anglès ASLR (*Address Space Layout Randomization*), traduït al català com "Aleatorietat en la disposició de l'espai de direccions". Aquesta opció de compilació està activada per defecte en la majoria de IDE i s'utilitza com a tècnica de seguretat informàtica, ja que el que fa és disposar de manera aleatòria les posicions de l'espai de direccions del procés, incloent

la direcció base, les posicions de la pila, el *heap*, les biblioteques, etc., de tal manera que a cada nova execució puguin contenir un valor diferent. Aquesta és una manera molt útil per complicar les enginyeries inverses que es centren en la recerca de dades, funcions i altra informació a memòria dels procesos, ja que cada instància diferent del mateix executable contindrà adreces diferents.

Les relocalitzacions són bàsicament adreces que han de ser arreglades en cas que un procés no pugui ser inicialitzat a la seva adreça base preferida (per defecte és l'adreça 0x00400000 pels executables PE [22]).

La part bona és que, en el cas que el nostre *stub* sigui compilat amb ASLR (on l'adreça base pot ser diferent a cada execució), i els possibles executables que puguin passar pel nostre *crypter* continguin funcions i dades que necessitin ser relocalitzades perquè el procés funcioni adequadament (en el cas que no pugui ser carregat a la seva adreça base preferida) apareixeran dins de la secció anomenada ***.reloc***.

Bàsicament, de manera resumida, el que hem de fer és calcular la diferència entre la direcció de la imatge base del nostre *malware* (l'obtenim també de les capçaleres del primer pas), i la direcció de la imatge base que hem obtingut al reservar espai quan hem fet el ***VirtualAllocEx*** en el pas 3 i, un cop tenim aquesta diferència, iterar per tota la secció **.reloc** suman-t'hi aquest valor [23, 24, 25].

6. Modificar els permisos del nou procés

Una cosa que no he mencionat abans, i he preferit fer-ho ara, és que quan es reserva amb ***VirtualAllocEx*** l'espai necessari pels bytes del *malware* al nou procés (pas 3), s'utilitza el flag *PAGE_EXECUTE_READWRITE* per evitar problemes de permisos. Tenir permisos de lectura, escriptura i execució per tot el procés pot arribar a ser sospitós de cara als antivirus i l'ideal seria assignar els permisos que contenen els bytes del *malware* original al nou procés un cop ja l'hem escrit al espai reservat per aquest.

Per fer això, utilitzarem la crida a l'API anomenada ***VirtualProtectEx*** [26] i ho farem tant per les capçaleres com per les seccions. Per les capçaleres afegirem només permisos de lectura (sempre ho són així per defecte) i per les seccions anirem iterant igual que ho hem fet al pas 4, buscant els *sectionHeader* (capçalera de cada secció) i, llegint el seu camp anomenat *Characteristics* [27], podrem saber quins permisos té la secció amb la que estem tractant en aquesta iteració.

7. Modificar la imatge base, el punt d'entrada i represa de l'execució del nou procés

Si bé s'ha mencionat al pas 2 que era necessari emmagatzemar el context del procés en una variable per a modificacions futures, ara, un cop que ja tenim els bytes del *malware* escrits al nou procés, la taula de relocalitzacions arreglada (en el cas que fos necessari) i els permisos modificats als originals, podem procedir a canviar el camp `EBX+8` [28] del context pel valor de la imatge base aconseguida del ***`VirtualAllocEx`*** del pas 3. Bàsicament, el registre `EBX + 8` bytes conté la imatge base de cada procés però, com que hem "desmapejat" el nostre procés per poder reservar un nou espai necessari pel *malware*, la imatge base resultant pot o no ser diferent, per tant és necessari realitzar aquesta modificació perquè el procés sàpiga quina és la seva nova pròpia imatge base. Hem de fer el mateix amb el registre `EAX` del context, però assignant-hi en aquest cas el camp ***`AddressOfEntryPoint`*** obtingut de les capçaleres del *malware*.

Un cop hem fet aquestes modificacions al context, podem tornar a assignar-lo al nostre procés amb la crida a l'API ***`SetThreadContext`*** [29] de tal manera que el nostre procés ja estigui preparat per ser executat (recordem que estava en estat suspès). La represa de l'execució la realitzem amb la crida a l'API ***`ResumeThread`*** [30].

De manera resumida doncs, els passos han estat:

1. Accedir a les capçaleres DOS i NT dels bytes que contenen el *malware*.
2. Crear un nou procés en estat suspès.
3. "Desmapejar" el seu contingut per poder reservar l'espai necessari pels bytes del *malware*.
4. Escriure aquests bytes al procés.
5. Fer les modificacions necessàries pel seu correcte funcionament (relocalitzacions).
6. Modificar els permisos de les capçaleres i seccions.
7. Assignar la nova direcció d'imatge base i punt d'entrada del procés i reprendre l'execució a memòria d'aquest nou procés que ara contindrà els bytes del *payload* maliciós, sense haver tocat en cap moment el disc dur.

6.1.2.2 Tècniques d'evasió d'antivirus

Arribats a aquest punt, podríem dir que ja tenim el nostre *crypter* implementat, perquè tenim els bytes del *malware* dins de l'executable de l'*stub* encriptats al disc dur i, quan aquest s'executa, es desencripten i s'injecten a memòria. Tot i així, això no és suficient per reduir les ràtios de detecció, ja que els antivirus analitzaran el nostre *stub* i intentaran trobar o bé signatures (que d'això no en trobaran, ja que s'ha programat des de zero i és únic), o mitjançant una anàlisi heurística, buscant patrons.

Recordem que el nostre *stub* conté una part que no està encriptada, que és la que s'encarregarà de desencriptar el *payload* i aplicar el mètode de *Process Hollowing*, per tant, l'antivirus podria analitzar aquests bytes i podria arribar a detectar certs patrons coneguts dins del món dels *crypters* (sobretot en la seqüència de crides a l'API de Windows que s'hagin realitzat, també anomenada *API Chain*).

Per combatre tot això, haurem d'aplicar algunes de les tècniques d'evasió existents. A continuació s'expliquen les més utilitzades.

Junk API Calls

Aquesta tècnica consisteix en afegir al codi crides aleatòries a l'API de Windows que no tinguin res a veure amb el que sol haver-hi a les implementacions comunes de *crypters* / *malware*. En el cas del nostre *crypter*, per exemple, es poden anar afegint aquestes crides entre pas i pas del mètode de *Process Hollowing*, de tal manera que podem trencar seqüències i patrons (o *API Chains*) heurístics coneguts pels antivirus amb la intenció de confondre'ls. En la Figura 6.3 es pot veure un exemple.

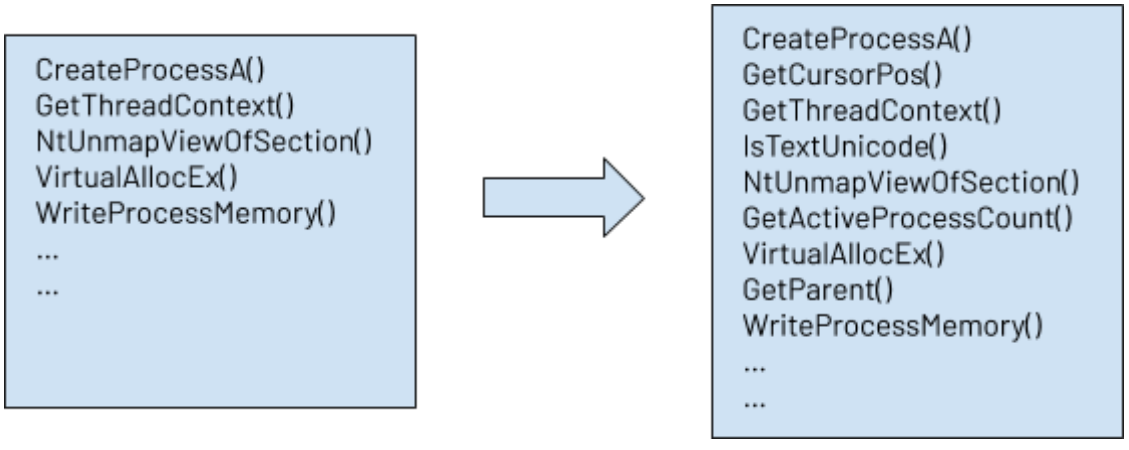

Figura 6.3 Al requadre esquerre, podem veure un exemple del que seria l'*API Chain* normal pel mètode de *Process Hollowing*. En canvi al dret, hi ha el mateix mètode de *Process Hollowing* però amb ***Junk API Calls*** intercalades, convertint-ho així en una ***API Chain*** diferent.

Realment, aquestes crides a l'API no són de cap utilitat per al nostre programa excepte el d'ajudar amb l'ofuscació.

Ofuscació d'importacions

Tot i haver-nos lliurat de que els antivirus puguin detectar patrons en les nostres *API Chains* afegint les *Junk API Calls*, alguns antivirus ja coneixen aquesta tècnica i simplement es limiten a comprovar quines són les crides a l'API de Windows que utilitza un programa en si. Si determinen que un programa està utilitzant certes crides a l'API molt populars entre el *malware* (realment, els *crypters* entren dins de la categoria de *malware*), poden arribar a marcar-lo com a maliciós tot i no haver trobat cap patró reconegut.

Cada crida a l'API que utilitzem s'anomena ***import*** dins del format del PE, i això és així perquè realment per poder utilitzar-les estem important la biblioteca de Windows fent un `#include <Windows.h>`, on aquesta, internament, té moltes més importacions. Aquests ***imports***, tal i com es menciona a l'apartat 1.6.3, estan emmagatzemats a la ***`Import Address Table`*** dins d'un executable i qualsevol antivirus pot llegir quines són les crides que s'estan utilitzant.

Si utilitzem un programa d'inspecció de fitxers PE per analitzar, per exemple, l'executable ***`Spotify.exe`***, podem veure totes les crides a l'API que s'han importat, sense necessitat d'executar-lo. La Figura 6.4 visualitza la inspecció d'un fitxer.

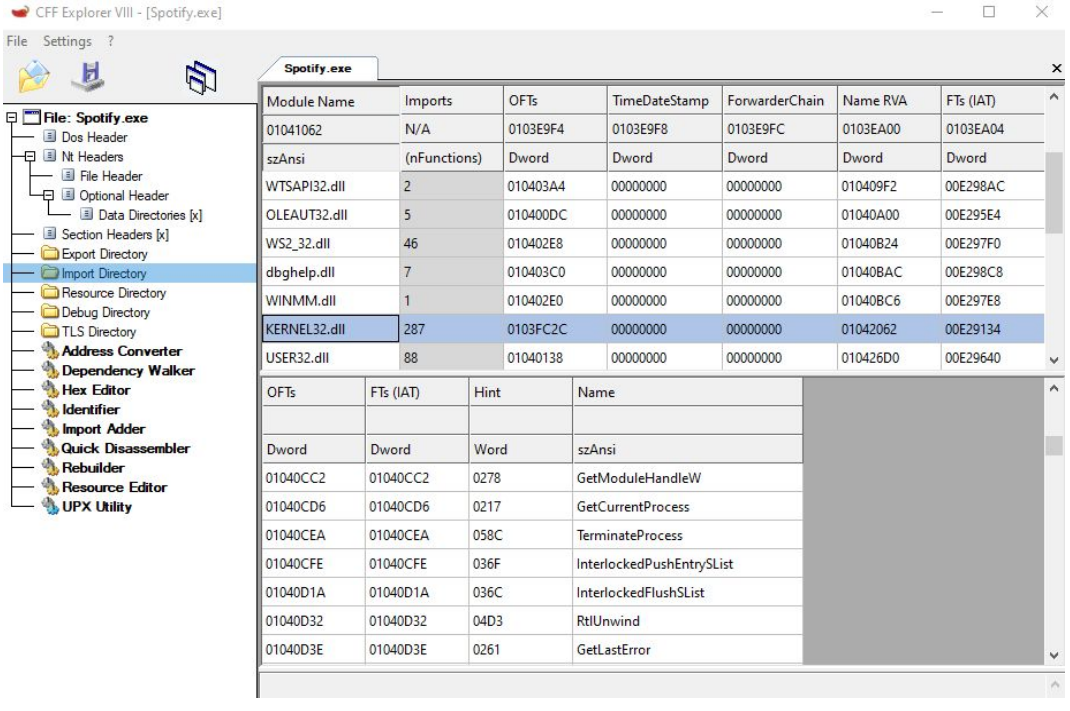

Figura 6.4 El programa CFF Explorer ens permet veure quines són les DLL que s'han importat en aquest executable i quines funcions, dins d'aquestes DLL, s'estan utilitzant.

Per evitar això, en comptes de cridar directament les funcions de l'API, podem anar-les a buscar a les seves DLL corresponents en temps d'execució mitjançant les crides a l'API ***GetModuleHandleA***[31] i ***GetProcAddress***[32]. Fins i tot podem anar més enllà i, en lloc d'utilitzar directament aquestes dues crides a l'API per trobar les altres que ens interessen, podem també anar-les a buscar recorrent l'estructura interna de les seves DLL. Inevitablement, haurem de fer servir la crida a l'API ***LoadLibraryA***[33] per poder carregar aquestes DLL en l'espai de memòria del nostre procés i poder recórrer-les, i aquest **import** serà impossible ofuscar-lo però, en realitat, també és molt comú trobar-lo inclòs en executables que no són *malware*. Per la funció ***GetModuleHandleA*** haurem de recórrer la DLL anomenada `ntdll.dll` i per la funció ***GetProcAddress*** la DLL `KERNEL32.DLL`. Realment, les DLL també segueixen el mateix format PE excepte algunes petites diferències amb els executables. La Figura 6.5 mostra el procés a seguir.

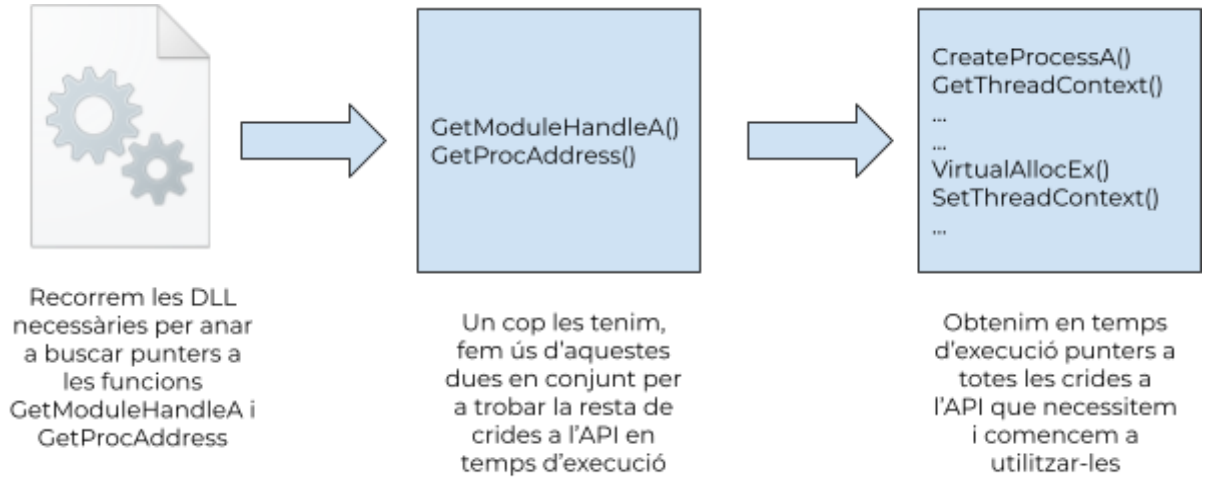

Figura 6.5 Passos del procés d'obtenció dels punters a les crides a l'API.

Òbviament, això només ho farem amb les crides a l'API que creiem que poden ser marcades com a sospitoses, en aquest cas les que s'utilitzen amb el mètode de *Process Hollowing*, però no ho farem, per exemple, per les *Junk API Calls* de l'apartat anterior, ja que de fet aquestes sí que ens interessa que apareguin en la ***Import Address Table*** del nostre *stub* (perquè són inofensives pel que fa a *malware*).

En la tècnica d'ofuscació d'importacions és on entren en acció els directoris ***IMAGE_DIRECTORY_ENTRY_EXPORT*** i ***IMAGE_DIRECTORY_ENTRY_IMPORT*** mencionats a l'apartat 1.6.3. Per aconseguir trobar els punters a les funcions ***GetModuleHandleA*** i ***GetProcAddress,*** haurem de recórrer la ***Export Table*** a la que apunta el camp `VirtualAddress` del directori ***IMAGE_DIRECTORY_ENTRY_EXPORT***. L'altre directori, com a tal, no ens serà d'utilitat a nivell de codi ja que no l'haurem de recórrer, però sí que hem de

saber que el seu camp `VirtualAddress` apunta a la ***Import Address Table*** descrita prèviament. Fent ús d'aquesta tècnica, no apareixerà llistada ninguna de les crides a l'API del mètode de *Process Hollowing* en aquesta taula, ja que aquestes, seran resoltes en temps d'execució.

Un altre punt a mencionar és que, un cop aconseguim els punters a les funcions ***GetModuleHandleA*** i ***GetProcAddress***, aquestes, si mirem la seva documentació [31,32], ens demanen com a paràmetres el nom de la DLL (de la que extreurem les funcions) i el nom de la funció en qüestió. Perquè el nostre *stub* no contingui strings literals fent referència a les funcions i les DLL, podem encriptar aquests paràmetres amb el mateix algoritme utilitzat per encriptar els bytes del *malware*, RC4 en aquest cas, i fer que es desencriptin els literals també en temps d'execució abans d'executar les crides a l'API que referencien. Això és necessari ja que tots els strings literals d'un executable s'emmagatzemen a la secció `.rdata`, i alguns antivirus també analitzen aquesta secció per intentar trobar strings sospitosos.

Anti-emulació de codi

Alguns antivirus intenten emular l'execució del codi dels programes per recopilar informació del comportament d'aquest sense comprometre el sistema operatiu de l'usuari. Això ho fan creant un entorn virtual, amb arxius ficticis, *registres de Windows*[3], DLL falses, etc., per veure amb què i com interactua l'executable en qüestió en aquest sistema. Per exemple, és molt comú que els *malwares* intentin modificar certs registres de Windows per comprometre l'ordinador de la víctima.

Per detectar un entorn d'emulació se'm va acudir fer ús de la crida a l'API ***LoadLibraryA***[33], també utilitzada en la tècnica anterior. Aquesta funció, bàsicament, carrega un mòdul (*DLL* en aquest cas) a l'espai de memòria del procés. La idea és intentar carregar o bé *DLL* inexistents amb noms inventats i veure si la funció ***LoadLibraryA*** no retorna NULL, o el cas contrari, intentar carregar *DLL* existents i comprovar si retorna NULL. Si es compleix alguna de les dues condicions, podem afirmar que el codi està sent emulat. Això és així ja que els entorns emulats dels antivirus disposen de recursos limitats i no poden emular tot el sistema i els seus fitxers. Quan hi ha crides com aquestes que busquen dependències externes, o bé retornen sempre un resultat qualsevol, existeixi o no, o bé no retornen res perquè no ho tenen emulat (per qüestió de recursos).

Aquesta funció que comprova si el codi està sent emulat serà la primera en executar-se dins de l'*stub*. En el cas que es detecti l'emulació de codi, el programa farà un ***return 0*** i l'*stub* es comportarà com si fos totalment

[3] El registre de Windows és una base de dades jeràrquica que emmagatzema paràmetres de configuració i opcions en els sistemes operatius de Windows.

inofensiu, sense arribar mai a desencriptar el *payload*. Si la funció no detecta cap emulació, l'*stub* seguirà la seva execució habitual.

6.2 Arquitectura del *SignatureClone*

Aquest és el tercer projecte i, realment, no forma part del *crypter* com a tal. Podria considerar-se una tècnica d'ofuscació addicional. Bàsicament, rebrà dos executables d'entrada i clonarà el certificat (o signatura digital) d'un executable a l'altre. No tots els executables tenen certificat però, els que provenen d'empreses legítimes que volen oferir software com a producte, solen obtenir certificats per als seus arxius. Anant a propietats d'un executable, podrem veure si aquest conté o no una firma digital. En la Figura 6.6 en podem veure un exemple del certificat digital del executable *Spotify.exe*:

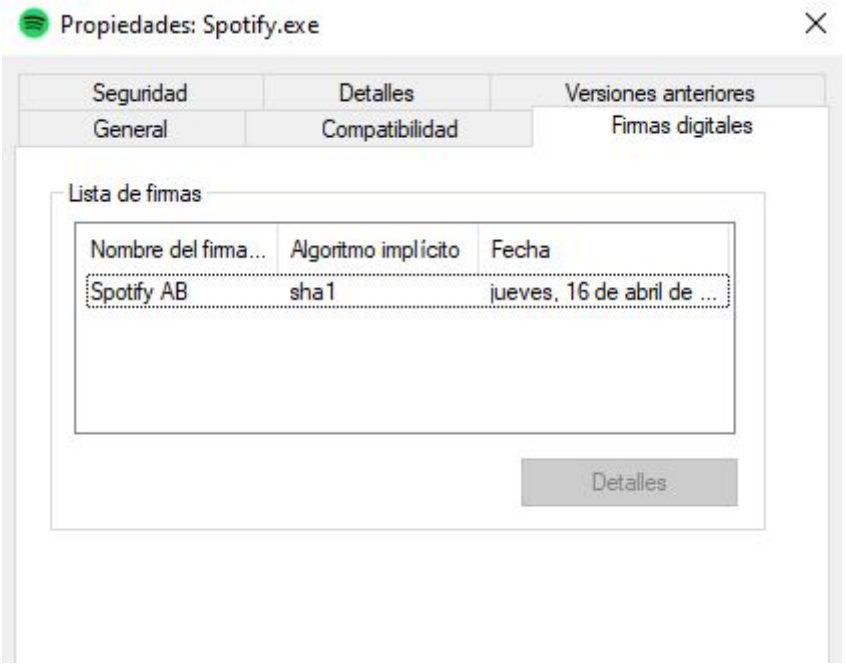

Figura 6.6 Firma digital de l'executable *`Spotify.exe`*.

No entrarem en detall en com s'aconsegueixen aquests certificats, ni com es generen, ja que realment això no ens interessa. El que sí que hem de saber és que alguns antivirus tenen en compte si un executable conté un certificat digital o no i, si no el conté, pot aixecar les sospites d'aquest cap a l'executable. Amb tot, això no vol dir que l'hagi de marcar com a maliciós només per això, ja que té en compte moltes altres coses que ja s'han anat explicant durant la documentació.

Com es menciona a l'apartat 1.6.3, aquests certificats digitals s'emmagatzemen dins de l'estructura interna del format PE, específicament a la direcció on apunta el camp ***`VirtualAddress`*** del directori de dades ***`IMAGE_DIRECTORY_ENTRY_SECURITY`***. Coneixent aquesta informació, haurem d'anar a buscar aquest directori a l'executable del que voldrem clonar el seu certificat, copiar tot el contingut de la seva taula fent ús dels camps ***`VirtualAddress`*** i ***`Size`*** del directori, i emmagatzemar aquesta còpia a l'executable final (el nostre *stub* que ja conté el *malware* encriptat en aquest cas), també recorrent l'estructura interna del PE per a trobar el seu ***`IMAGE_DIRECTORY_ENTRY_SECURITY`*** corresponent i saber on emmagatzemar la firma digital copiada.

6.3 Arquitectura de la *Webapp*

La *webapp* és bastant senzilla i s'ha fet a mode de presentació per no estar utilitzant la consola del sistema de Windows (més conegut com a `CMD`). També ha estat d'utilitat per a totes les proves que he hagut de realitzar, ja que és més fàcil utilitzar la web que estar escrivint constantment les ordres amb els arguments a la consola cada cop que volgués fer passar un arxiu pel meu *crypter*, per després clonar-li a dins una signatura d'algun executable, etc.

Aquest projecte, com quasi bé totes les *webapps*, es divideix en un *frontend* i un *backend*.

6.3.1 *Frontend* de la *webapp*

El *frontend* serà un simple formulari que l'usuari haurà de completar en el que se li demana el següent:

- L'arxiu que vol que passi pel *crypter* (el *malware* en aquest cas).
- La clau d'encriptació a utilitzar (amb un botó amb opció d'auto-generar, que crea automàticament una clau de 32 caràcters alfanumèrics aleatoris).
- La icona que apareixerà a l'arxiu resultant.
- L'arxiu executable del que l'usuari voldrà clonar el certificat digital.
- El nom del fitxer resultant.

Quan l'usuari ha completat el formulari, podrà prémer el botó "*CRYPT FILE!*" per fer *submit* de les dades que ha introduït al formulari. Aquestes aniran al *backend*, on l'executable passarà pel *crypter*, se li modificaran la icona i el certificat digital, i serà retornat al *frontend* en una pestanya nova

donant-li l'opció a l'usuari de descarregar el nou executable indetectable pels antivirus.

Interfície de la *webapp*

La *webapp* consta de dues pantalles principals, amb un fons dinàmic (que lògicament no es pot apreciar en una imatge). En la Figura 6.7 es pot apreciar la interfície que es mostra inicialment.

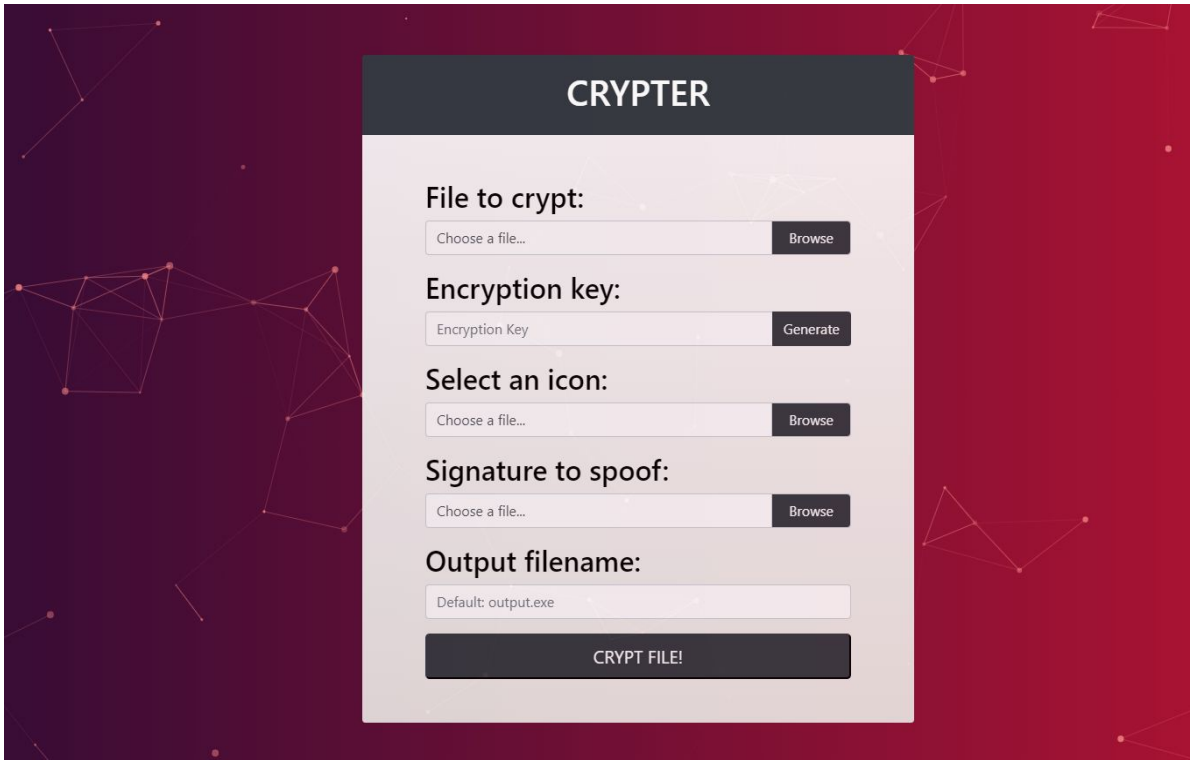

Figura 6.7 Interfície que mostra la pantalla principal quan l'usuari obre la *webapp*.

Quan l'usuari omple el formulari amb tota la informació necessària descrita anteriorment, podrà prémer el botó "***CRYPT FILE!***", i aquest botó canviarà de color a verd, es modificarà el contingut del text amb un *spinner* i un missatge per avisar a l'usuari que el seu arxiu està sent processat i que pot durar uns segons. En la Figura 6.8 es pot apreciar aquesta modificació del botó.

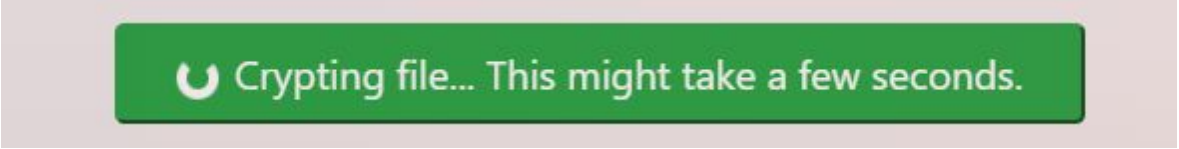

Figura 6.8 Botó no clicable que indica a l'usuari que l'arxiu seu està passant pel *crypter* i està rebent altres modificacions (icona, certificat digital i el nom de sortida).

En la Figura 6.9 es mostra la interfície final que li apareixerà a l'usuari un cop l'arxiu final ja està llest, indicant-li ja que pot descarregar el seu nou executable.

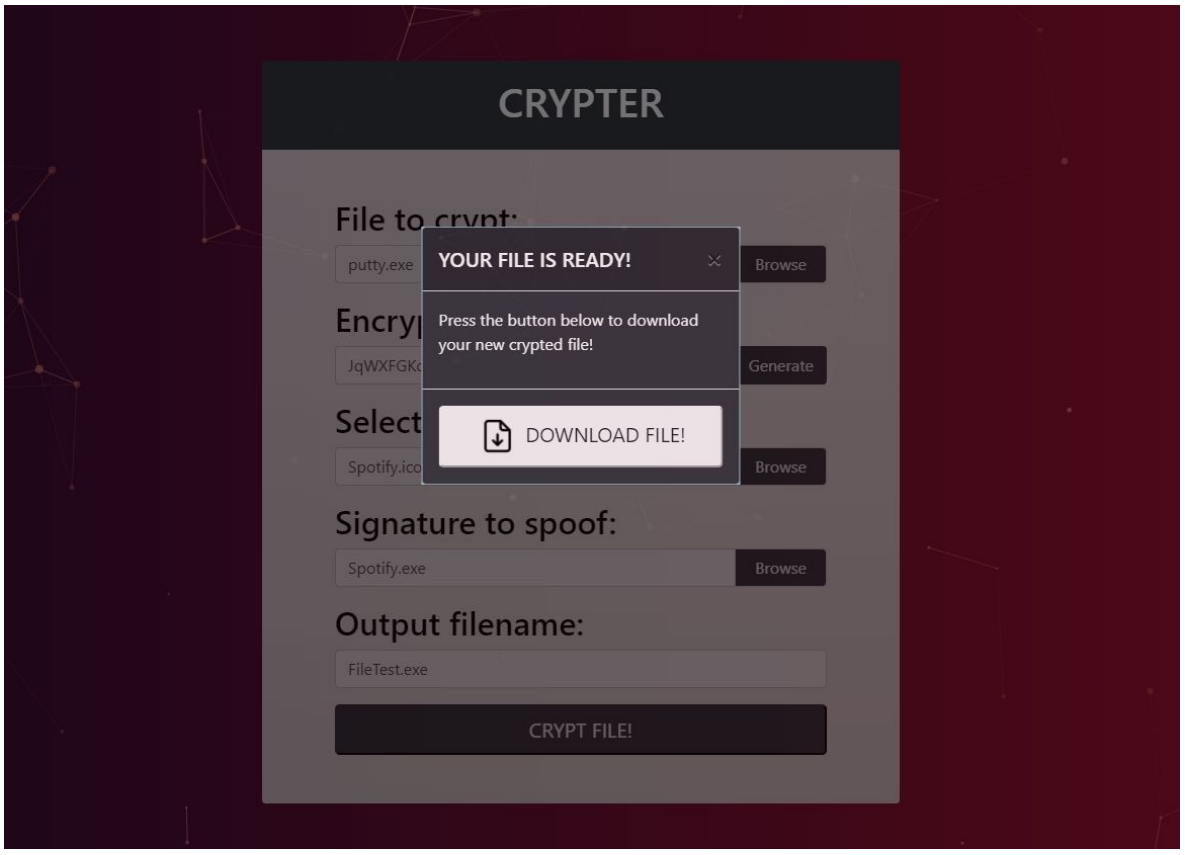

Figura 6.9 Interfície final on apareix el fons amb una opacitat reduïda i una nova finestra al centre indicant a l'usuari que ja pot descarregar el seu executable final.

Si l'usuari prem el botó "***DOWNLOAD FILE!***", l'arxiu serà descarregat automàticament al seu navegador. En la Figura 6.10 hi podem trobar un exemple.

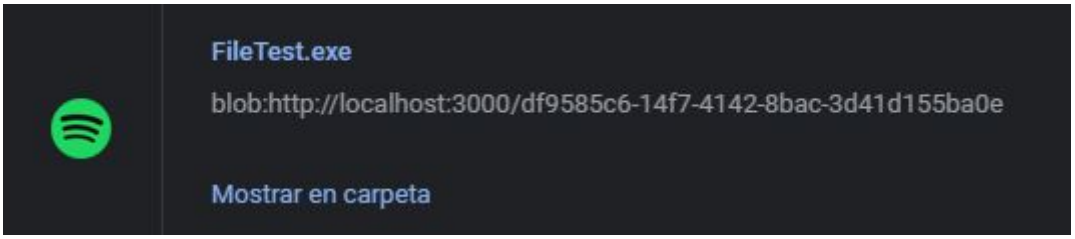

Figura 6.10 Descàrrega final del fitxer resultant en el navegador de l'usuari. Aquest fitxer conté els bytes del executable *putty.exe* encriptats i el certificat digital i la icona de *Spotify.exe*.

6.3.2 *Backend* de la *webapp*

El *backend* serà molt senzill ja que aquest projecte no consta de cap gestió d'usuaris ni volums de dades i, per tant, no conté cap base de dades. El *backend* ens servirà per generar automàticament el fitxer resultant sense que l'usuari pugui veure què és el que està passant darrere i fer-ho molt més còmode. El *backend* consta de dos *endpoints* on:

- El primer *endpoint* és un mètode **POST**[4] i és l'encarregat d'agafar totes les entrades de dades del formulari i processar els arxius rebuts per generar l'executable resultant. L'ordre serà el següent:

    1) L'executable rebut en el camp "`File to crypt`" passarà pel ***builder***, serà encriptat usant la clau d'encriptació del camp "`Encryption key`", i es generarà un nou arxiu `.h` amb els bytes d'aquest executable encriptats.
    2) Aquest arxiu `.h` serà copiat a la carpeta del projecte de l'***stub***, junt a la icona introduïda en el camp "`Select an icon`".
    3) El projecte de l'***stub*** serà compilat i es generarà un nou arxiu executable.
    4) Utilitzant el projecte ***SignatureClone***, es copiarà el certificat digital de l'executable del camp "`Signature to spoof`" cap a l'executable generat per l'***stub***.
    5) Es modificarà el nom del fitxer final pel del camp "`Output filename`".

- El segon *endpoint* és un mètode **GET**[5] i és l'encarregat de retornar aquest fitxer resultant a l'usuari, que podrà descarregar directament prement el botó "***`DOWNLOAD FILE!`***".

[4] El mètode POST d'un *endpoint* s'utilitza per enviar dades a un servidor per crear/actualitzar un recurs.
[5] El mètode GET d'un *endpoint* s'utilitza per sol·licitar dades a un servidor sobre un recurs específic.

7. Implementació

En aquesta secció s'explicarà com s'ha estructurat a nivell d'implementació tot el conjunt del projecte. Es descriuran les tecnologies utilitzades, es mostrarà l'arbre de fitxers de cada projecte i s'explicarà què hi ha implementat en cada un d'aquests arxius. Es pot trobar tot el codi al repositori de ***GitHub*** [34].

La Figura 7.1 mostra l'estructura del projecte en el repositori de GitHub.

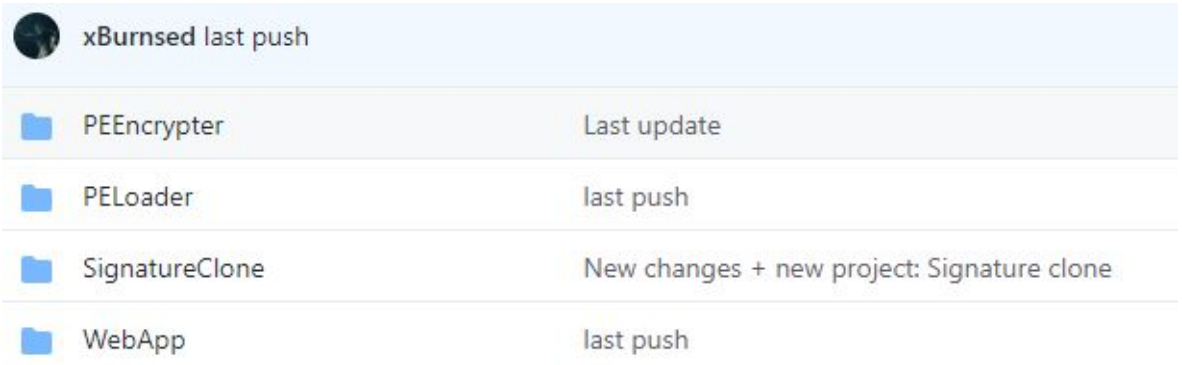

Figure 7.1 Repositori GitHub del projecte dividit en quatre carpetes. Cada una d'aquestes conté la seva implementació corresponent segons la seva finalitat.

7.1 *PEEncrypter*

Aquest projecte correspon al *Builder* del *crypter* i ha estat programat en C++ utilitzant el Visual Studio 2019 com a IDE. La Figura 7.2 mostra l'estructura de fitxers d'aquest projecte.

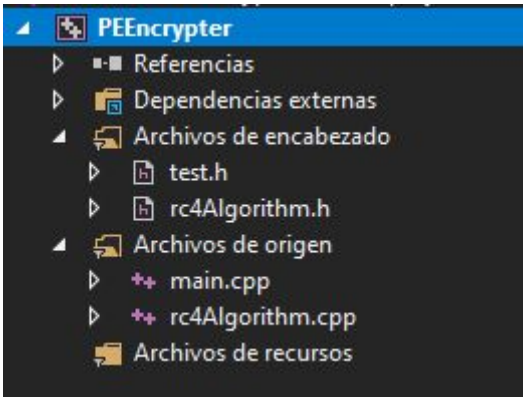

Figura 7.2 Estructura de fitxers del *Builder* en el IDE Visual Studio 2019.

A continuació es descriuen breument els fitxers que conformen el *Builder*.

1. `main.cpp:` Conté la part principal del codi amb el que, utilitzant l'algoritme **RC4**, s'encriptaran els bytes del fitxer executable que rebi com a argument i els codificarà en **Base64**. Després crearà un nou fitxer de sortida anomenat **`shellc.h`** que contindrà aquests bytes.

2. `rc4Algorithm.h i rc4Algorithm.cpp:` Capçalera i implementació del algoritme d'encriptació **RC4**.

3. `test.h`: Arxiu que s'ha utilitzat per realitzar proves i verificar que la implementació de l'algoritme d'encriptació **RC4** es correcta.

7.2 *PELoader*

Aquest projecte correspon a l'*Stub* del crypter i és el més important dels quatre, doncs és la base i nucli d'aquest. La implementació ha estat programada amb C++ utilitzant Visual Studio 2019 com a IDE. La Figura 7.3 mostra l'estructura de fitxers d'aquest projecte.

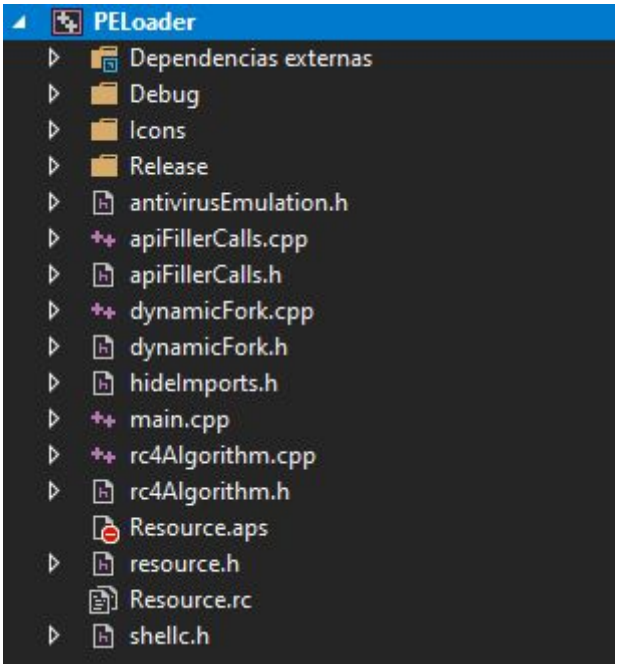

Figura 7.3 Estructura de fitxers de l'*stub* en el IDE Visual Studio 2019.

A continuació es descriuen breument els fitxers que conformen l'*stub*:

1. `main.cpp:` Arxiu que conté la part principal del codi. S'encarregarà de decodificar i posteriorment desencriptar amb l'algoritme **RC4** els bytes que conté l'arxiu `shellc.h`. Després injectarà a memòria la imatge d'aquests bytes utilitzant el mètode de ***Process Hollowing*** junt amb les tècniques d'ofuscació descrites anteriorment.

2. `rc4Algorithm.h i rc4Algorithm.cpp:` Capçalera i implementació de l'algoritme d'encriptació **RC4**.

3. `dynamicFork.h i dynamicFork.cpp:` Capçalera i implementació del mètode d'injecció a memòria ***Process Hollowing*** (també conegut com a *Dynamic Forking*).

4. `antivirusEmulation.h:` Aquesta capçalera conté la implementació per detectar els entorns d'emulació dels antivirus.

5. `hideImports.h:` Capçalera que conté la implementació de la tècnica d'ofuscació d'importacions.

6. `apiFillerCalls.h i apiFillerCalls.cpp:` Capçalera i implementació de les *Junk API Calls* utilitzades com a tècnica d'ofuscació.

7. `shellc.h`: Arxiu que serà reemplaçat dinàmicament pel *backend* de la *webapp* cada cop que l'usuari vulgui fer passar pel *crypter* un nou executable. Conté els bytes encriptats en **RC4** i codificats en **Base64** d'aquest.

8. `resource.h i resource.rc:` Arxius que contenen les configuracions necessàries perquè el projecte compilat utilitzi la icona de la carpeta "`Icons`".

7.3 SignatureClone

Per aquest petit projecte no hi ha una imatge representativa de la estructura de fitxers, ja que consta d'un únic arxiu que conté tota la implementació, i és el propi `main.cpp.`

També ha estat programat amb C++ utilitzant Visual Studio 2019.

El `main.cpp` conté la funció responsable de retreure en quina posició es troba el certificat digital i la mida d'aquest dins d'un executable (en cas

que en tingui), per a posteriorment extreure-ho i guardar-ho en una variable, perquè pugui ser copiat a un altre executable nou (el nostre).

7.4 *WebApp*

La *webapp* consta de dues implementacions distintes utilitzant diferents llenguatges de programació i *frameworks*[6]. Com s'ha descrit en l'apartat de disseny, la *webApp* està dividida en dues parts: el *frontend* i el *backend*.

7.4.1 *Frontend*

Per al *frontend*, s'ha utilitzat ReactJS, una biblioteca Javascript de codi obert dissenyada per crear interfícies d'usuari amb l'objectiu de facilitar el desenvolupament d'aplicacions web. S'ha utilitzat l'editor de text Visual Studio Code.

La Figura 7.4 mostra l'estructura de fitxers del *frontend*.

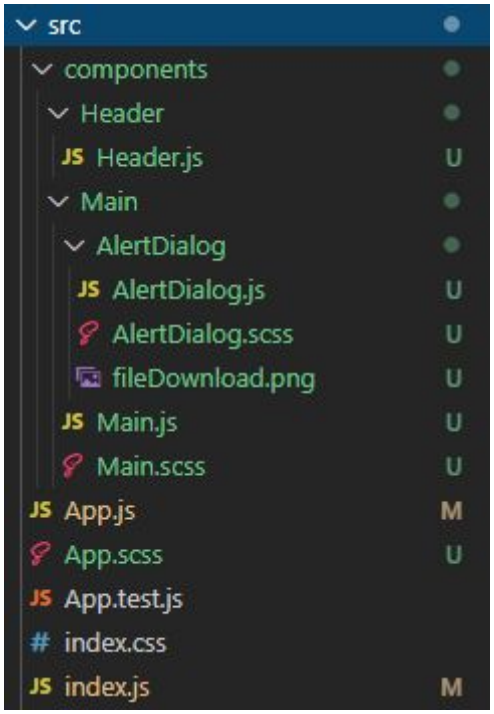

Figura 7.4 Estructura de fitxers de ReactJS pel *frontend* en l'editor de text Visual Studio Code.

[6] Un framework (entorn de treball) és una infraestructura de programari que, en la programació orientada a objectes, facilita la concepció de les aplicacions mitjançant la utilització de biblioteques de classes o generadors de programes.

A continuació es descriuen breument els fitxers que conformen el *frontend*:

1. `index.js i index.css:` Arxius base. Contenen la instanciació de la *webapp* de *Reactjs* amb una fulla d'estils genèrics.

2. `App.js i App.scss:` Contenen el *render* dels components base que formaran la web i els estils que s'aplicaran de manera externa a aquestos (posicionament). Aquests components es troben a la carpeta "`components`".

3. `Header.js:` Conté el codi pel títol de la finestra central principal on hi ha escrit "`CRYPTER`".

4. `Main.js i Main.scss:` Contenen el codi del formulari principal, on hi apareixen tots els *inputs* necessaris perquè l'usuari pugui penjar els fitxers i introdueixi altres paràmetres per obtenir l'arxiu final després de passar pel *crypter*.

5. `AlertDialog.jss i AlertDialog.scss:` Contenen el codi de la finestra que apareix per descarregar l'arxiu final després de fer passar l'executable pel *crypter*.

7.4.2 *Backend*

Per al *backend*, s'ha utilitzat Flask, un framework de Python minimalista que permet crear aplicacions web amb el mínim de línies de codi possible. S'ha utilitzat l'editor de text Visual Studio Code. La Figura 7.5 mostra l'estructura de fitxers del *backend*.

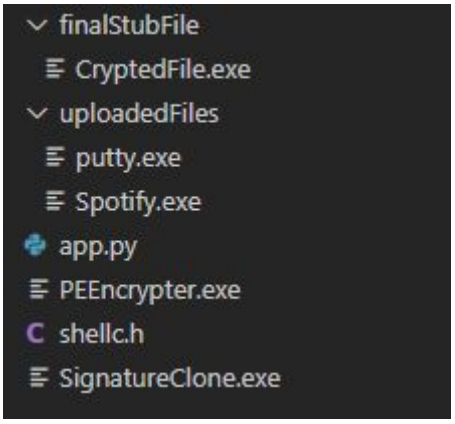

Figura 7.5 Estructura de fitxers de ReactJS pel *frontend* en l'editor de text Visual Studio Code.

A continuació es descriuen breument els fitxers que conformen el *backend*:

1. `app.py:` Arxiu que conté la implementació del *backend* i les definicions dels seus respectius *endpoints*.

2. `PEEncrypter.exe:` Executable del projecte del *Builder*. Utilitzant el *backend* s'encripta l'executable penjat per l'usuari i es genera l'arxiu `shellc.h`.

3. `shellc.h:` Arxiu que conté els bytes encriptats de l'executable penjat per l'usuari, generat després d'executar el `PEEncrypter.exe`. Aquest serà posteriorment copiat a la carpeta que conté el projecte de l'*Stub*, i es realitzarà una compilació d'aquest des del *backend*.

4. `SignatureClone.exe:` Executable del projecte *SignatureClone*. Utilitzant el *backend,* es copia un certificat digital d'un executable penjat per l'usuari a l'arxiu generat després de la compilació del projecte de l'*Stub* amb el `shellc.h,` que conté els bytes encriptats, convertint-lo així en l'arxiu final. Aquest arxiu final rebrà el nom indicat per l'usuari des del formulari del *frontend*.

5. Carpeta `uploadedFiles:` Carpeta on es guardaran els executables penjats per l'usuari: l'arxiu a fer passar pel *crypter* i l'arxiu del que clonar la signatura. `putty.exe` i `Spotify.exe`, en són exemples, respectivament.

6. Carpeta `finalStubFile:` Carpeta que contindrà l'arxiu final que ja ha passat pel *crypter* i conté la icona, el nom i el certificat digital desitjats per l'usuari. El fitxer contingut en aquesta carpeta serà el fitxer que descarregarà l'usuari des del seu navegador.

8. Model d'avaluació

Per analitzar l'eficàcia del nostre *crypter* s'ha utilitzat el servei d'anàlisi d'antivirus online **AntiScan.Me** [8]. El servei AntiScan.Me permet analitzar simultàniament i de manera molt senzilla arxius executables amb un total dels 26 antivirus comercials més coneguts en busca d'amenaces, obtenint els resultats d'aquestes anàlisis molt ràpidament. S'ha escollit el servei d'anàlisi d'AntiScan.Me sobre altres *multiscanners* perquè ofereix fins a sis escaneigs gratuïts per IP, no distribueix els resultats a les empreses dels antivirus (evitant així anàlisis exhaustives i manuals) i ofereix l'opció d'obtenir els resultats en format imatge.

L'arxiu que s'ha utilitzat per realitzar les proves és un *malware* anomenat **Remcos** [35]. Aquest tipus de *malware* és un RAT (*Remote Administration Tool*), un tipus de virus molt potent que otorga control absolut a l'atacant de l'ordinador de la víctima. En la Figura 8.1 es poden apreciar algunes d'aquestes funcions, com el veure en directe la pantalla de la víctima, espiar la *webcam*, *keylogger* (un *log* per robar contrasenyes i demés informació de la víctima), *shell*[7] remota, controlar i accedir els arxius de l'ordinador de l'usuari, etc.

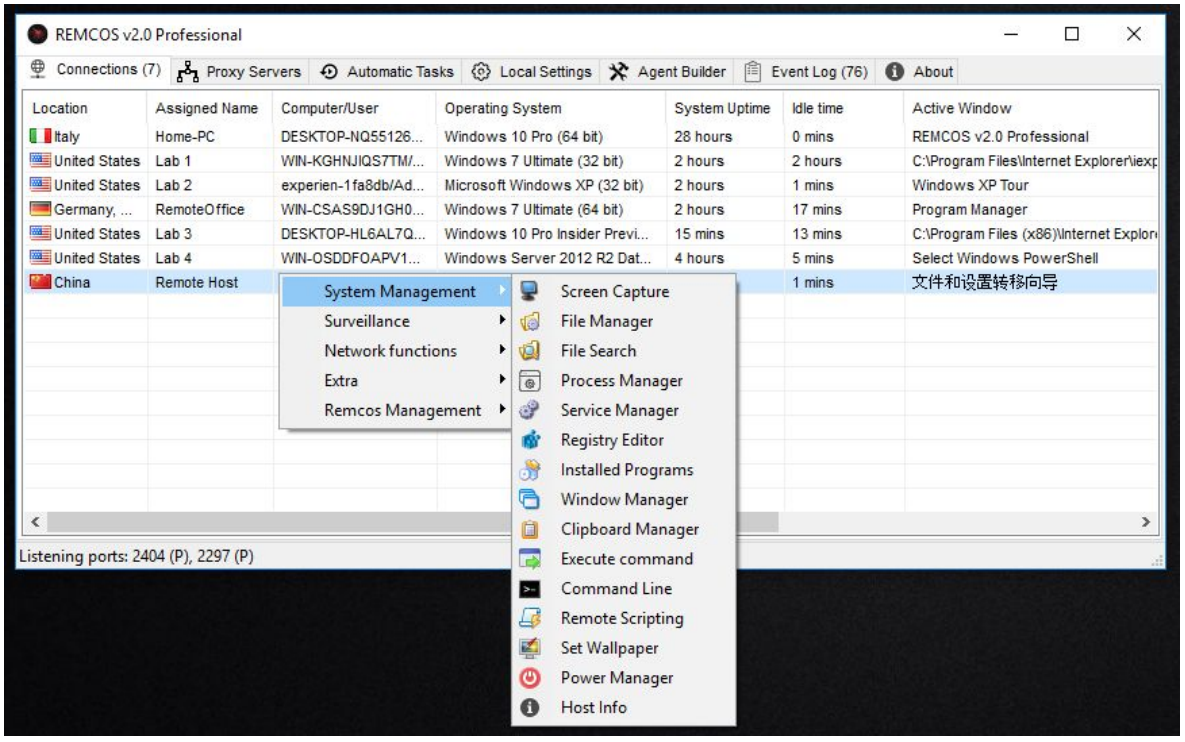

Figura 8.1 Panell principal del RAT Remcos on es poden apreciar les víctimes a les que té accés i algunes de les funcions que pot realitzar [35].

[7] Programa d'ordinador que ofereix accés al sistema operatiu i serveis d'aquest a través d'un intèrpret d'ordres (consola del sistema de Windows).

La primera anàlisi serà amb l'arxiu *malware* intacte, és a dir, sense passar pel nostre *crypter* ni cap altra eina. Els resultats es poden apreciar en la Figura 8.2. Del total de 26 antivirus que ofereix aquesta pàgina, podem veure com aquest *malware* és detectat, quan no ha passat pel *crypter*, per un total de **23** antivirus. Alguns inclús reconeixen quin tipus de *malware* és i l'etiqueten com a **Remcos**. Això ens indica que la majoria d'aquests antivirus tenen actualment constància d'aquest arxiu i han generat signatures per la detecció d'aquest. Podem veure com inclús el mateix Windows Defender, antivirus que ve instal·lat per defecte amb el sistema operatiu de Windows, detecta aquest arxiu com a maliciós; això vol dir que, tot i que l'usuari no tingui instal·lat cap antivirus comercial al seu ordinador, aquest *malware* serà automàticament detectat i eliminat del seu sistema.

Figura 8.2 Resultats obtinguts de l'anàlisi realitzada pel servei d'anàlisi d'antivirus online AntiScan.Me amb el *malware* Remcos.

8.1 Test inicial

Primerament, es valorarà l'eficàcia del *crypter* utilitzant únicament la tècnica d'injecció a memòria de ***Process Hollowing*** (és el mínim requeriment essencial), sense l'ús de cap tècnica d'ofuscació descrita en l'apartat **6.1.2.2.** En la Figura 8.3 es poden apreciar els resultats obtinguts en aquesta anàlisi. Sembla ser que s'ha aconseguit una reducció de **23**/26 deteccions a **8**/26 que, tot i ser bastant, aquest *malware* no es pot considerar FUD encara. Això ens confirma que tot i que l'arxiu *stub* contingui els bytes del *malware* encriptats en disc dur, els antivirus no només busquen signatures sinó que mitjançant anàlisis heurístiques busquen patrons genèrics, miren quines crides a l'API s'han utilitzat, si conté o no certificat digital, executen l'arxiu en un entorn emulat, etc.

Figura 8.3 Resultats obtinguts de l'anàlisi realitzada pel servei d'anàlisi d'antivirus online AntiScan.Me amb el *malware* Remcos després de passar pel *crypter* i sense utilitzar cap tècnica d'ofuscació addicional.

8.2 Segon test

El segon test ha estat utilitzant el mateix arxiu anterior, sense cap tècnica d'ofuscació addicional exceptuant el **Process Hollowing** essencial del *crypter*, però utilitzant el projecte *SignatureClone* per copiar el certificat digital de l'aplicació `Spotify.exe`. En la Figura 8.4 podem apreciar els resultats d'aquest test. D'aquesta manera hem pogut passar de **8**/26 deteccions a **5**/26, indicatiu que la cosa va millorant però encara no és suficient perquè aquest *malware* sigui completament indetectable a ulls dels antivirus.

Figura 8.4 Resultats obtinguts de l'anàlisi realitzada pel servei d'anàlisi d'antivirus online AntiScan.Me amb el *malware* Remcos després de fer-lo passar pel *crypter* amb el certificat digital de l'aplicació `Spotify.exe` clonat.

8.3 Tercer test

En aquest test s'ha utilitzar el *crypter* amb les tècniques d'ofuscació relacionades amb les crides a l'API: *Junk API Calls* i ofuscació d'importacions. No s'ha utilitzat ni el clonatge del certificat digital ni la tècnica d'anti emulació. En la Figura 8.5 es poden apreciar els resultats obtinguts. Podem concloure aquí que cadascun dels antivirus té el seu propi criteri d'avaluació i ponderen certs valors de manera diferent. Alguns li donen més importància al fet que un arxiu no contingui certificat digital mentre d'altres es fixen més en les crides a l'API que s'han utilitzat i els patrons d'aquestes.

Figura 8.5 Resultats obtinguts de l'anàlisi realitzada pel servei AntiScan.Me amb el *malware* Remcos després de fer-lo passar pel *crypter* amb les tècniques d'ofuscació relacionades amb l'API de Windows: ***Junk API Calls*** i ofuscació d'importacions.

8.4 Quart test

En aquest test s'ha utilitzat el *crypter* amb les tècniques de l'apartat anterior (ofuscació d'importacions i Junk API Calls) més el clonatge del certificat digital de la aplicació `Spotify.exe`. No s'ha utilitzat la tècnica d'anti emulació encara. En la figura 8.6 es poden apreciar els resultats obtinguts. Sembla ser, que l'antivirus propi de Windows ha estat l'únic capaç de detectar el *malware* (això dona molt que pensar!). No sabem amb profunditat com funcionen els entorns d'emulació i és possible que els resultats variessin si s'utilitzés un altre tipus de *malware*. Una de les meves hipòtesis és que, en termes d'emulació, l'antivirus de Windows és el més potent ja que disposa d'accés directe als recursos del seu propi sistema operatiu.

Figura 8.6 Resultats obtinguts de l'anàlisi realitzada pel servei AntiScan.Me amb el *malware* Remcos després de fer-lo passar pel *crypter* amb les tècniques d'ofuscació relacionades amb l'API de Windows (***Junk API Calls*** i ofuscació d'importacions) i el clonatge del certificat digital `Spotify.exe`.

8.5 Test final

En el test final s'utilitzarà el *crypter* juntament amb totes les tècniques d'ofuscació addicionals implementades i el certificat digital de l'arxiu `Spotify.exe` clonat per al mateix *malware* de ***Remcos***. Els resultats es poden apreciar en la Figura 8.7. Utilitzant totes les tècniques junt amb el *crypter* hem pogut passar de **23**/26 deteccions a **0**/26, transformant aquest *malware* en un arxiu completament indetectable (*FUD*).

Figura 8.7 Resultats obtinguts de l'anàlisi realitzada pel servei AntiScan.Me amb el *malware* Remcos després de fer-lo passar pel *crypter* juntament amb totes les tècniques d'ofuscació addicionals i el clonatge del certificat digital de l'aplicació `Spotify.exe`.

9. Conclusions i treball futur

Aquest projecte ens ha ajudat a entendre en detall com funcionen internament els executables de Windows, gràcies a l'estudi necessari del format PE que ens ha calgut realitzar per poder implementar el *crypter*. Hem après varies tècniques d'ofuscació destinades a l'evasió dels antivirus i una tècnica d'injecció a memòria, com és el ***Process Hollowing***, tot i que que tenim constància que hi ha més tècniques d'ofuscació i mètodes d'injecció a memòria que es poden incorporar. Aquest coneixement adquirit sobre els *crypters*, de per si, ja ens és útil per donar-nos compte que l'usuari mig mai està protegit del tot, encara que aquest usi antivirus.

Una de les conclusions personals, i suposició al mateix temps, que he tret realitzant moltes proves (no totes documentades aquí), és que els antivirus, quan no disposen d'una signatura de l'arxiu que estan analitzant, per determinar si aquest és maliciós o no utilitzen com una espècie de sistema de puntuació, on van sumant punts si van veient coses que no els agrada. Si l'arxiu aconsegueix arribar a certs punts, és marcat com a maliciós. Imagino que aquest és un dels motius perquè a vegades es poden arribar a crear falsos positius. Per exemple, és possible que alguns antivirus, si veuen que un arxiu no té icona o no té certificat digital o fa servir certa crida de l'API de Windows, etc. sumi punts, tot i que potser no sigui un arxiu maliciós.

A nivell personal, tot i que esperava poder reduir la ràtio de detecció d'un executable *malware* actualment detectat pels antivirus, en un cert percentatge, no esperava poder arribar a modificar-lo per aconseguir un *malware FUD*, ja que ho miris com ho miris, aquestes companyies disposen de molts recursos i diners, i al principi això era bastant intimidant. Per tant, puc afirmar que estic molt satisfet amb els resultats i el projecte ha superat les meves expectatives inicials.

De cara al treball futur, hi ha moltes coses que poden ser millorades i afegides al *crypter*. La primera a considerar seria el poliformisme. Si s'aconsegueix fer el codi del *crypter* polimòrfic, on a cada generació de l'*stub* per al mateix arxiu executable el contingut binari fos diferent però la funció final la mateixa, no ens preocuparia en absolut que es generessin signatures pel nostre *stub*. De fet, part del *crypter* es podria dir que ja és polimórfica, perquè els bytes encriptats del *malware* que conté l'*stub*

poden haver estat encriptats utilitzant una clau diferent per al mateix executable. Però, la part que realment ens interessa convertir en polimòrfica és la que no està encriptada i conté el mètode d'injecció a memòria i altres tècniques d'ofuscació addicionals.

També seria interessant millorar la GUI de la *webapp* afegint-hi més opcions com, per exemple, donar a escollir a l'usuari entre diferents mètodes d'injecció a utilitzar pel seu *stub,* afegir també compatibilitat per a executables d'arquitectura x64, afegir-hi un *binder*[8] (també com a *proof of concept*), poder clonar arxius *manifest*[9], etc.

En general, de manera resumida, el fet d'estudiar el funcionament intern dels *crypters* i haver-ne implementat un, ens ha servit com a base per seguir aprenent i desenvolupant sobre temes relacionats, per tal d'entendre encara millor com funciona el món de la seguretat informàtica en Windows i, en particular, el funcionament del *malware* sobre aquest sistema operatiu.

[8] Un *binder* és un programa que rep d'entrada dos executables i els uneix en un de sol. En aquest context per exemple, es podria passar pel *binder* el nostre stub que conté el *malware* junt amb una aplicació legitima com `Postman.exe`, de tal manera que quan la víctima executi l'arxiu vegi una execució normal del `Postman` però per darrere s'estigués executant el *malware* sense que aquesta sospités re.

[9] Els arxius *manifest* contenen metadades que descriuen el nom del software, número de versió, nom de l'empresa, llicència, descripció del producte, etc. La majoria de software legítim sol tenir un arxiu *manifest*.

Bibliografia

[1] Riot Games - *RIOT'S APPROACH TO ANTI-CHEAT.*
[Consulta: 6 d'Octubre 2020]
https://technology.riotgames.com/news/riots-approach-anti-cheat

[2] Wikipedia - *Software Cracking.*
[Consulta: 6 d'Octubre 2020]
https://en.wikipedia.org/wiki/Software_cracking

[3] Xiao-bin Wang, Guang-yuan, Yang Yi-chao Li Dan Liu - *Review on the application of Artificial Intelligence in Antivirus Detection System.*
[Consulta: 20 d'Octubre 2020]
http://vigir.missouri.edu/~gdesouza/Research/Conference_CDs/IEEE_CyberIntSys_2008/PDFFILES/Papers/P0183.pdf

[4] Joxean Koret, Elias Bachaalany - *The Antivirus Hacker's Handbook.*
[Consulta: 21 d'Octubre 2020]

[5] Dr. Mafaz Mohsin Khalil Al-Anezi - *Generic Packing Detection using Several Complexity Analysis for Accurate Malware Detection.*
[Consulta: 30 d'Octubre 2020]

[6] Vitani - *Spartan Crypter GOLD | Supports C++ & .NET | RT & ST FUD.*
[Consulta: 30 d'Octubre 2020]
https://hackforums.net/showthread.php?tid=5946671

[7] *Antivirus and Cybersecurity Statistics, Trends & Facts 2020.*
[Consulta: 3 de Novembre 2020]
https://www.safetydetectives.com/blog/antivirus-statistics/

[8] Antiscan.me - *Online Virus Scanner Without Result Distribution*
[Consulta: 5 de Novembre 2020]
https://antiscan.me/

[9] Wikipedia - *Iterative and incremental development.*
[Consulta: 15 de Novembre 2020]
https://en.wikipedia.org/wiki/Iterative_and_incremental_development

[10] Iterative incremental Development Models.
[Consulta: 15 de Novembre 2020]
https://istqbfoundation.wordpress.com/2017/09/18/iterative-incremental-development-models/

[11] Glassdoor - *Sous mitjans de Barcelona segons lloc de treball.*
[Consulta: 25 de Novembre 2020]
https://www.glassdoor.es/Sueldos/barcelona-sueldo-SRCH_IL.0,9_IM1015.htm?clickSource=searchBtn

[12] Wikipedia - *RC4.*
[Consulta: 5 de Desembre 2020]
https://en.wikipedia.org/wiki/RC4
https://www.dcode.fr/rc4-cipher

[13] LoadResource function - *Win32 API.*
[Consulta: 5 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadresource

[14] Wikipedia - *Base64.*
[Consulta: 5 de Desembre 2020]
https://es.wikipedia.org/wiki/Base64

[15] Docs Microsoft - *PE HEADERS.*
[Consulta: 15 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_nt_headers32
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_file_header
https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header32

[16] Docs Microsoft - *CreateProcessA.*
[Consulta: 15 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

[17] Docs Microsoft - *GetThreadContext*.
[Consulta: 15 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getthreadcontext

[18] Vergilius Project - *PEB*.
[Consulta: 15 de Desembre 2020]
https://www.vergiliusproject.com/kernels/x86/Windows%2010/1909%2019H2%20(November%202019%20Update)/_PEB

[19] NTAPI Undocumented Functions - *NtUnmapViewOfSection*.
[Consulta: 20 de Desembre 2020]
http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FSection%2FNtUnmapViewOfSection.html

[20] Docs Microsoft - *VirtualAllocEx*.
[Consulta: 20 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualallocex

[21] Docs Microsoft - *WriteProcessMemory*.
[Consulta: 23 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory

[22] Docs Microsoft - *Optional Header Windows-Specific Fields (Image Only)*
[Consulta: 27 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#optional-header-windows-specific-fields-image-only

[23] Wikipedia - *Aleatoriedad en la disposición del espacio de direcciones*.
[Consulta: 27 de Desembre 2020]
https://es.wikipedia.org/wiki/Aleatoriedad_en_la_disposici%C3%B3n_del_espacio_de_direcciones

[24] Stack Exchange - *How do packers/crypters deal with ASLR?*
[Consulta: 27 de Desembre 2020]
https://security.stackexchange.com/questions/225293/how-do-packers-crypters-deal-with-aslr

[25] Docs Microsoft - *The .reloc section.*
[Consulta: 27 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#the-reloc-section-image-only

[26] Docs Microsoft - *VirtualProtectEx.*
[Consulta: 30 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotectex

[27] Docs Microsoft - *Section Header. Characteristics.*
[Consulta: 30 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#section-table-section-headers

[28] Stack Overflow - *Windows initial execution context.*
[Consulta: 30 de Desembre 2020]
https://stackoverflow.com/questions/6028849/windows-initial-execution-context

[29] Docs Microsoft - *SetThreadContext.*
[Consulta: 30 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setthreadcontext

[30] Docs Microsoft - *ResumeThread.*
[Consulta: 30 de Desembre 2020]
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread

[31] Docs Microsoft - *GetModuleHandleA.*
[Consulta: 5 de Gener 2021]
https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea

[32] Docs Microsoft - *GetProcAddress.*
[Consulta: 5 de Gener 2021]
https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

[33] Docs Microsoft - *LoadLibraryA*.
[Consulta: 5 de Gener 2021]
https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

[34] GitHub - *Poc-Packer*.
[Consulta: 10 de Gener 2021]
https://github.com/xBurnsed/PoC-Packer

[35] BLEEPINGCOMPUTER - *Cybercriminals Undeterred by ToS For Remcos RAT*.
[Consulta: 15 de Gener 2021]
https://www.bleepingcomputer.com/news/security/cybercriminals-undeterred-by-tos-for-remcos-rat/

[36] MITRE - *Process Injection*
[Consulta: 20 de Gener 2021]
https://attack.mitre.org/techniques/T1055/

[37] Elastic - *Ten process injection techniques: A technical survey of common and trending process injection techniques*
[Consulta: 20 de Gener 2021]
https://www.elastic.co/es/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

ANNEX

Tècniques d'injecció de codi a memòria

En aquest projecte s'ha explicat en detall la tècnica d'injecció a memòria escollida: el **Process Hollowing**. És important destacar que hi ha d'altres alternatives, ni millors ni pitjors, que ens poden proporcionar la mateixa funció. En aquest annex es cobriran per sobre alguns d'aquests mètodes, i podrem veure que molts procediments són similars excepte petites diferències que els caracteritzen [36, 37].

DLL Injection

En aquesta tècnica el procés del *malware* crea un arxiu DLL que conté el *payload* (codi maliciós) i, al mateix temps, aquest procés va a buscar un altre procés qualsevol (i a poder ser, legítim), per reservar espai virtual per la DLL aquesta, i s'assegura que aquest procés remot executi aquesta DLL maliciosa creant-hi un nou *thread*.

PE Injection

Aquesta tècnica és molt similar a l'anterior però el procés del *malware* no crea ninguna DLL al disc dur (el que és un avantatge), sinó que injecta el *payload* maliciós directament. El procediment és similar: a) el procés del *malware* reserva espai al procés remot; b) es crea un nou *thread* en el procés remot; c) s'escriu el contingut del *malware* en *shellcode* (conjunt de bytes); d) es realitzen les modificacions necessàries perquè el procés executi aquest nou codi injectat, e) s'executa el nou *thread*.

Un podria pensar també que és molt semblant al **Process Hollowing** que hem utilitzat nosaltres, però la diferència clau és que en el cas de **PE Injection** s'està injectant codi en un procés remot (i no en una còpia del mateix procés) i que en aquesta tècnica no es "desmapeja" l'espai de memòria actual del procés sinó que es reserva espai, es crea un *thread* remot, s'escriu el contingut, es realitzen modificacions d'adreces base i altres adreces essencials i s'executa el nou *thread*.

Process Doppelgänging

En aquest mètode s'utilitza el sistema *Transaction NTFS* que ofereix Windows per realitzar transaccions entre arxius de manera segura. Aquesta tècnica és de les més avançades i possiblement de les més complicades. La part positiva és que al contrari de la tècnica **Process Hollowing**, aquest mètode no fa ús de les crides a l'API altament monotoritzades com `NtUnmapViewOfSection`, `VirtualProtectEx`, i `SetThreadContext`, entre d'altres. El funcionament és el següent:

1. Es crea una nova transacció per a un executable legítim (per exemple, el `svchost.exe` de Windows).
2. Es sobreescriu aquest arxiu amb el codi maliciós.
3. Es crea una nova secció per aquest arxiu que apunti a aquest executable maliciós.
4. Es realitza un *rollback* de la transacció inicial esborrant d'aquesta manera els canvis realitzats a l'arxiu executable legítim inicial, però mantenint la secció creada amb el contingut maliciós.
5. Es crea un nou procés (amb el seu *thread* principal) utilitzant aquesta secció que conté el *malware*.
6. S'executa el nou procés maliciós.

Thread Execution Hijacking

Tècnica també molt similar a la de **Process Hollowing** però en aquest cas no es crea cap procés nou sinó que es va a buscar un procés existent. La diferència entre aquest mètode i el de **PE Injection** prèviament descrit, és que en aquest sí que es desmapeja l'espai de memòria del procés remot en el que es vol injectar el codi maliciós, mentre que en el de **PE Injection** no (cosa que implica que s'han de fer reajustaments d'adreces degut al nou codi injectat).

Bàsicament el funcionament és el següent:

1. Es va a buscar un procés existent.
2. Es pausa el *thread* principal d'aquest procés.
3. Es "desmapeja" el seu contingut i es reserva un nou espai pels bytes del *malware*.
4. S'escriuen els bytes del *malware*.
5. Es reprèn l'execució del *thread*.

Diagrama de Gantt

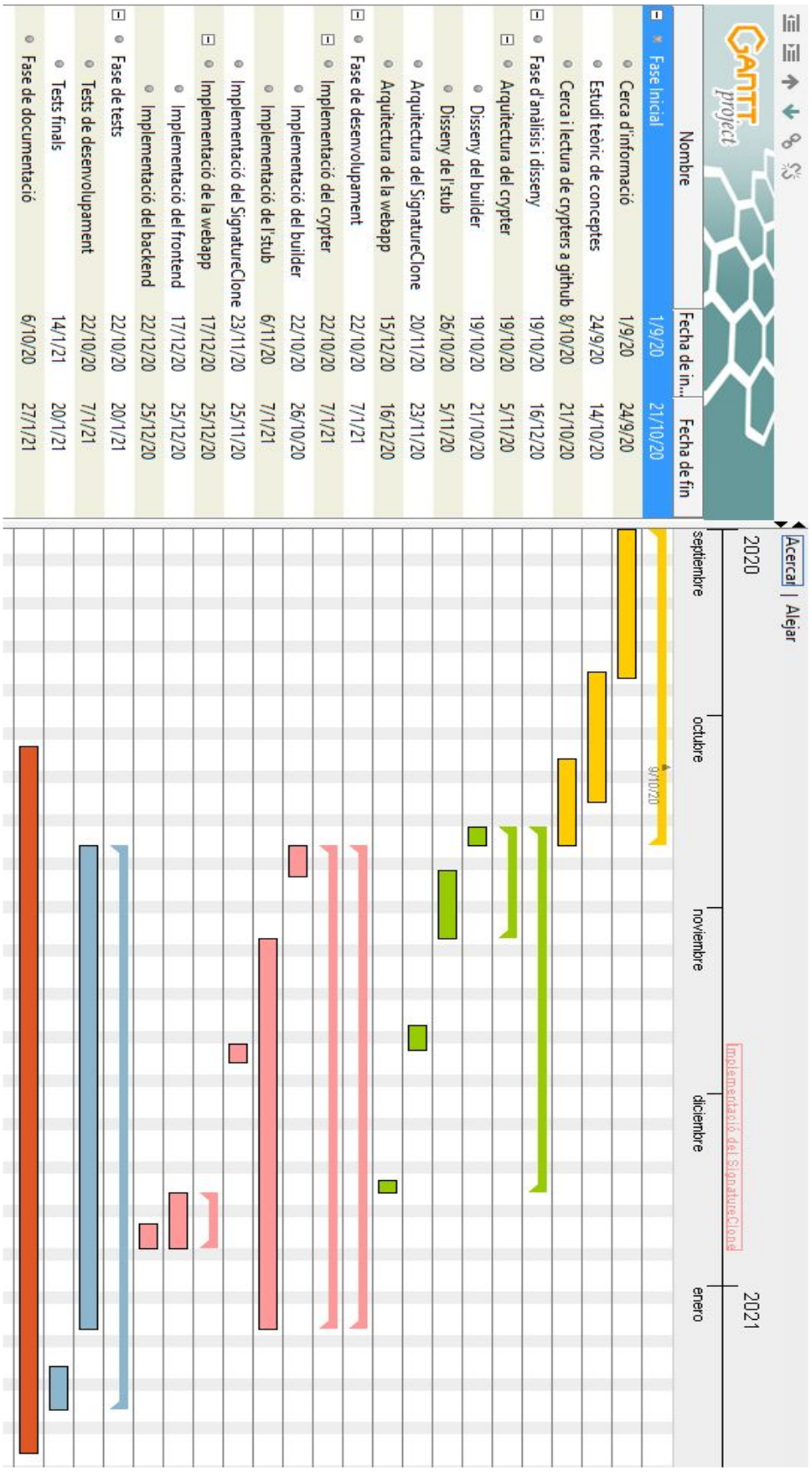

Diagrama de Gantt ampliat.