ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ

ΕΦΑΡΜΟΓΩΝ ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ

«ΑΝΤΙΣΤΡΟΦΗ ΜΗΧΑΝΙΚΗ ΛΟΓΙΣΜΙΚΟΥ (REVERSE CODE ENGINEERING)»

ΦΟΙΤΗΤΗΣ: **ΤΖΙΩΤΖΗΣ ΓΑΒΡΙΗΛ**

ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ: **02/2091**

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: **ΒΑΦΕΙΑΔΗΣ ΑΝΤΩΝΗΣ**

ΘΕΣΣΑΛΟΝΙΚΗ 2011

Πρόλογος

Η παρούσα πτυχιακή εργασία πραγματοποιήθηκε στο Τμήμα Πληροφορικής του Αλεξάνδρειου Τεχνολογικού Εκπαιδευτικού Ίδρυματος Θεσσαλονίκης. Στόχος της είναι η μελέτη της επιστήμης της Αντίστροφης Μηχανικής Λογισμικού (Reverse Code Engineering).

Ειδικότερα, γίνεται παρουσίαση τεχνικών οι οποίες υιοθετούνται από διαρρήκτες λογισμικού (crackers) στην προσπάθεια τους να επέμβουν σε λογισμικό (για το λειτουργικό σύστημα των Windows) το οποίο δεν τους ανήκει καθώς και των τεχνικών που χρησιμοποιούνται από προγραμματιστές εμπορικών εφαρμογών στην προσπάθεια τους να προστατευτούν. Ακόμα γίνεται αναφορά στον θετικό αντίκτυπο που μπορεί να έχει η ΑΜΛ στον χώρο της πληροφορικής αλλά και τα ηθικά ζητήματα που προκύπτουν από την κακή πρακτική της επιστήμης αυτής.

Περίληψη

Η ΑΜΛ - αντίστροφη μηχανική λογισμικού ή αποσυμπίληση λογισμικού (ReverseCodeEngineering) χρησιμοποιείται για να υποστηρίξει την κατανόηση του λογισμικού εκμεταλλευόμενη ως βασική πηγή πληροφόρησης για την οργάνωση και συμπεριφορά του λογισμικού τον εκτελέσιμο κώδικα.

Στα πλαίσια της εργασίας γίνεται παρουσιάζονται βασικές έννοιες που αφορούν την ΑΜΛ όπως η συμβολική γλώσσα (Assembly) και οι καταχωρητές καθώς και η σημασία τους στην ΑΜΛ. Αναλύεται η δομή των εκτελέσιμων αρχείων των Windows τα οποία αποκαλούνται φορητά εκτελέσιμα και περιγράφονται από το πρότυπο PE-COFF (Portable Executable – Common Object File Format) της Microsoft. Παρουσιάζονται οι βασικές τεχνικές της ΑΜΛ, δηλαδή η ανακατασκευή αντικειμενικού κώδικα (Disassembly), η απομεταγλώττιση (Decompilation) και η αποσφαλμάτωση (Debugging). Ακόμα παρουσιάζονται μερικά από τα χρησιμότερα εργαλεία της ΑΜΛ τα οποία στη συνέχεια χρησιμοποιούνται κατά την αντιστροφή της λειτουργίας εφαρμογών.

Επίσης παρουσιάζονται τεχνικές που χρησιμοποιούν οι μηχανικοί λογισμικού ώστε να κατανοήσουν αλλά και να επέμβουν κακόβουλα στον κώδικα του εκτελέσιμου αρχείου. Αυτές περιλαμβάνουν την ανίχνευση (Tracing), την εύρεση των σημείων ενδιαφέροντος μέσα στον κώδικα, την απλή και την προχωρημένη επιδιόρθωση κώδικα και τέλος τη δημιουργία γεννητριών κλειδιών μέσα από την αντίστροφη αναγνώριση αλγορίθμων.

Τέλος παρουσιάζεται η λογική πίσω από τεχνικές οι οποίες χρησιμοποιούνται κυρίως σε εμπορικές εφαρμογές για την προστασία των πνευματικών δικαιωμάτων των δημιουργών δυσχεραίνοντας την αντιστροφή των εφαρμογών τους όπως η αντι-αποσφαλμάτωση, εφαρμογές συμπίεσης (packers) και προστασίας (protectors) του εκτελέσιμου αρχείου, έλεγχοι κώδικα κυκλικού πλεονασμού, έλεγχοι μέσω διακομιστών και κρυπτογράφηση ονομάτων μεθόδων και συμβολοσειρών.

Περίληψη στα Αγγλικά (Abstract)

RCE – Reverse Code Engineering or simply Reverse Engineering (RE) is the science of understanding software code by using the executable code as the only source of information.

In this paper, basic concepts of RCE are presented, such as the Assembly Language and computer registers as well as their meaning for RCE. The structure of Windows' executable files also called as portable executables is analyzed as it is defined in Microsoft's PE-COFF (Portable Executable – Common Object File Format) standard. The basic methods of RCE, Disassembly, Decompilation and Debugging are presented along with some of the most useful tools used to reverse engineer applications.

Furthermore is explained how Reverse Engineers employ their knowledge to understand the inner working of a software application and modify it's behavior (sometimes with malignant purposes).The techniques involve tracing, effectively locating points of interest in the code, simple and advanced patching of code and finally creating a key generator for an application through reverse engineering entire algorithms.

Finally, the logic behind various techniques used to protect the legal rights of application creators focused on preventing or making reverse engineering of an application harder are presented. These include anti-debugging tricks as well as complete applications that pack or protect the executable. Cyclic Code Redundancy Checks (CRC Checks), server checks code obfuscation (for strings and method names) is also explained.

Ευχαριστίες

Θέλω αρχικά να ευχαριστήσω τον επιβλέπων καθηγητή μου κ. Βαφειάδη Αντώνιο ο οποίος μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το πολύ ενδιαφέρον και σημαντικό θέμα καθώς και για το χρόνο του και τις συμβουλές του που με καθοδήγησαν ώστε να παρουσιάσω το θέμα όσο το δυνατόν καλύτερα και πιο κατανοητά. Στην προσπάθεια μου να εμβαθύνω στον τομέα της ΑΜΛ ουκ ολίγες φορές χρειάστηκε να ανατρέξω στις σημειώσεις του από τα μαθήματα πάνω στη συμβολική γλώσσα και την αρχιτεκτονική των υπολογιστών. Οι βασικές αυτές αρχές που διδάχθηκα από τον κ. Βαφειάδη ήταν καίριες για να μπορέσω να κατανοήσω τον δύσκολο χώρο της επιστήμης της ΑΜΛ.

Θέλω επίσης να ευχαριστήσω τον αδερφό μου Χρήστο Τζιώτζη ο οποίος με την εμπειρία του στην συγγραφή εργασιών με βοήθησε να οργανώσω καλύτερα τις ιδέες μου και να τις αποτυπώσω στη συνέχεια στην παρούσα εργασία.

Τέλος θέλω να ευχαριστήσω την οικογένεια μου που μου παρείχε όλες τις δυνατές διευκολύνσεις ώστε να μπορέσω να ασχοληθώ αποκλειστικά με την εργασία μου όπως και για την συμπαράσταση τους.

Περιεχόμενα

Ευρετήριο σχημάτων

Ευρετήριο πινάκων

Ευρετήριο τμημάτων κώδικα

1. Εισαγωγή

Η ΑΜΛ - αντίστροφη μηχανική λογισμικού ή αποσυμπίληση λογισμικού (Reverse Code Engineering) χρησιμοποιείται για να υποστηρίξει την κατανόηση του λογισμικού εκμεταλλευόμενη ως βασική πηγή πληροφόρησης για την οργάνωση και συμπεριφορά του λογισμικού τον εκτελέσιμο κώδικα. Ακόμα θα μπορούσαν να εξάγονται αλληλοσυμπληρωματικά διαγράμματα τα οποία παρέχονται στους προγραμματιστές ως εναλλακτικές παρουσιάσεις των πληροφοριών. Επίσης με την περαιτέρω ανάλυση του εκτελέσιμου κώδικα και καθώς παρουσιάζονται οι πληροφορίες σε υψηλότερο επίπεδο ο προγραμματιστής υιοθετεί άλλες προοπτικές. Τέλος, δίνεται η δυνατότητα προσανατολισμού στη δομή, στη συμπεριφορά ή στις εσωτερικές καταστάσεις των αρχείων.

Η αντίστροφη μηχανική λογισμικού έχει σημαντικό ρόλο στον κύκλο ζωής του λογισμικού και ιδιαίτερα στην εξέλιξη ενός ήδη υπάρχοντος συστήματος. Τόσο η κατανόηση του λογισμικού όσο και η ανάλυση των επιδράσεων μπορούν να υποστηριχτούν από γνώσεις που προέρχονται από την ΑΜΛ.

Ένα άλλο σημαντικό στοιχείο είναι πως η φάση της συντήρησης μιας εφαρμογής λογισμικού έχει αποδειχτεί ως η πιο χρονοβόρα και μεγαλύτερη σε κόστος. Ενδεικτικά, το 90% του κόστους του λογισμικού σε μια επιχείρηση προκύπτει από την συντήρηση του. Κατά την φάση αυτή το λογισμικό αλλάζει και βελτιώνεται συνεχώς. Πολλές φορές και κυρίως σε συστήματα που έχουν αναπτυχθεί πολλά χρόνια πριν, πηγές πληροφοριών όπως η τεκμηρίωση και τα διαγράμματα είναι ατελείς, δεν υπήρξαν ποτέ ή δεν υπάρχουν πια. Τέλος με την ραγδαία αύξηση στην χρήση του αντικειμενοστραφούς προγραμματισμού και λόγω των επιδράσεων του δυσχεραίνεται ακόμα περισσότερο η συντήρηση με μόνη πηγή πληροφοριών τον πηγαίο κώδικα. Για παράδειγμα, όπως γνωρίζουμε η συμπεριφορά μιας αντικειμενοστραφούς εφαρμογής λογισμικού προκύπτει κυρίως από τις αλληλεπιδράσεις μεταξύ των αντικειμένων. Οι σχετικές εντολές μπορεί να διανέμονται σε πολλές κλάσεις, οι οποίες αναθέτουν σχεδόν όλες τις λειτουργίες τους σε άλλες κλάσεις ενώ διαχειρίζονται πολύ λίγη από την συνολική δουλειά. Δηλαδή, σε πολλές εφαρμογές δεν υπάρχει συνοχή στις κλάσεις και η σύζευξη είναι πολύ αυξημένη. Σε αυτό το σημείο διαγράμματα που προκύπτουν από την ΑΜΛ μπορούν να παρουσιάσουν τέτοιες

συνεργασίες κλάσεων – αντικειμένων και να βοηθήσουν στην κατανόηση των εσωτερικών λειτουργιών της εφαρμογής.

Πέρα όμως από την συντήρηση λογισμικού η ΑΜΛ επιδρά δραστικά και στην καθημερινή ζωή του χρήστη ο οποίος βλέπει άμεσα τα αποτελέσματα της στο λογισμικό προστασίας από κακόβουλο λογισμικό. Ο μόνος τρόπος κατανόησης της συμπεριφοράς κακόβουλου λογισμικού και εξουδετέρωσης του είναι μέσα από τις διάφορες τεχνικές της ΑΜΛ.

Σε αυτό το σημείο όμως να επισημάνουμε πως η ΑΜΛ δεν είναι κάτι που μπορεί να γίνει από ένα άτομο μόνο. Συνήθως απαιτούνται αρκετά άτομα και ένας σημαντικός αριθμός υποστηρικτικών προγραμμάτων για να μπορέσει να επιτευχθεί ο τελικός στόχος.

Ο τελικός αυτός στόχος μπορεί να παρουσιαστεί σε πολλές διαφορετικές μορφές όπως:

- Δημιουργία τεκμηρίωσης.
- Αποσφαλμάτωση λογισμικού.
- Αλλαγή συμπεριφοράς του λογισμικού σε ορισμένες καταστάσεις.
- Προσθήκη λειτουργικότητας για την οποία δεν είχε σχεδιαστεί αρχικά το λογισμικό.
- Αναπαραγωγή λειτουργικότητας σε καινούριο λογισμικό.

Πολύ ενδιαφέρον παρουσιάζει όμως και η ερώτηση «Γιατί να ασχοληθεί κάποιος με την ΑΜΛ αν δεν σχετίζεται άμεσα με το αντικείμενο της εργασίας του;». Η πραγματικότητα είναι πως υπάρχουν δεκάδες προσωπικοί λόγοι που θα μπορούσαν να υποκινήσουν κάποιον να μάθει περισσότερα. Συνήθως κάθε πραγματικός οπαδός της πληροφορικής θέλει να μαθαίνει πως δουλεύει το αγαπημένο του λογισμικό. Να μπορεί να κατανοήσει την κάθε του λειτουργία. Να μπορεί να το αλλάξει κατά βούληση ώστε να ανταπεξέρχεται καλύτερα στις δικές του ανάγκες. Πολλές φορές ειδικά, όταν οι εταιρείες σταματούν την υποστήριξη για κάποιες εφαρμογές τους και ο χρήστης δεν μπορεί να ελπίζει σε διόρθωση των προβλημάτων της εφαρμογής ή σε προσθήκη νέων λειτουργιών.

Άλλες φορές πάλι δυστυχώς το κίνητρο είναι η παρανομία. Όπως κάθε είδος γνώσης και η ΑΜΛ μπορεί να χρησιμοποιηθεί από επιτήδειους με χαρακτηριστικό παράδειγμα την κλοπή πνευματικής ιδιοκτησίας. Οι crackers προσπαθούν να επιδείξουν

τις ικανότητες τους στοχεύοντας εμπορικές εφαρμογές προστασίας λογισμικού. Τεράστιος πόλεμος διεξάγεται καθημερινά μεταξύ των εταιριών και της πειρατείας.

Η αρνητική πλευρά της ΑΜΛ έχει οδηγήσει σε τεράστιες αντιπαραθέσεις σχετικά με την νομιμότητα της χρήσης τέτοιων τεχνικών αν και δεν υπάρχει σαφώς ορισμένο νομικό πλαίσιο. Απλουστευμένα θα μπορούσαμε να περιγράψουμε ως παράνομη χρήση της ΑΜΛ την οποιαδήποτε προσπάθεια κλοπής πνευματικής ιδιοκτησίας.

Το μόνο σίγουρο είναι πως μαθαίνοντας περισσότερα για την ΑΜΛ απαραίτητη είναι η γνώση της συμβολικής γλώσσας (assembly) και ο μηχανικός λογισμικού (Reverse Engineer) που ασχολείται με αυτόν τον τομέα ακόμα και αν δεν γνωρίζει τίποτα, σύντομα θα είναι σε θέση να κατανοεί τα πάντα γύρω από την συμβολική γλώσσα. Τέλος, με αυτόν τον τρόπο βελτιώνει τις προγραμματιστικές του ικανότητες και την αποδοτικότητα του καθώς αποκτά την δυνατότητα να σκέφτεται σε χαμηλό επίπεδο, κατανοώντας το επίπεδο στο οποίο δουλεύει ο υπολογιστής.

2. Η συμβολική γλώσσα (Assembly)

Εισαγωγή

Αρχικά να αναφέρουμε πως η Assembly αποτελεί μια συμβολική αναπαράσταση της γλώσσας μηχανής του επεξεργαστή. Η αναγκαιότητα για τη δημιουργία της συμβολικής γλώσσας αυτής προέκυψε από το γεγονός πώς είναι πιο εύκολο για έναν άνθρωπο να θυμάται συμβολικά ονόματα εντολών παρά τις αντίστοιχες εντολές στο 16δικό σύστημα. Έτσι για παράδειγμα είναι πιο εύκολο να θυμάται κάποιος την εντολή «pop» η οποία επιστρέφει την τελευταία τιμή από τον σωρό (stack) αντί του αντίστοιχου κώδικα λειτουργίας (opcode) σε γλώσσα μηχανής «1F». Κάθε δεδομένο που μπορεί να αποκωδικοποιηθεί ονομάζεται εντολή (instruction) και το σύνολο τους (instruction set) αποτελεί μια συλλογή όλων των δυνατών λειτουργιών που μπορεί να εκτελέσει ο επεξεργαστής. Ο συμβολομεταφραστής (assembler) είναι το πρόγραμμα που στη συνέχεια αναλαμβάνει τη μετάφραση του κώδικα της συμβολικής γλώσσας σε γλώσσα μηχανής. Για να το επιτύχει αυτό αντικαθιστά τα διάφορα μνημονικά (mnemonics) της συμβολικής γλώσσας σε κώδικες λειτουργίας καθώς και τους τελεστέους (operands) με την καθαρή τους δυαδική μορφή, αναγνώσιμη από τον επεξεργαστή.

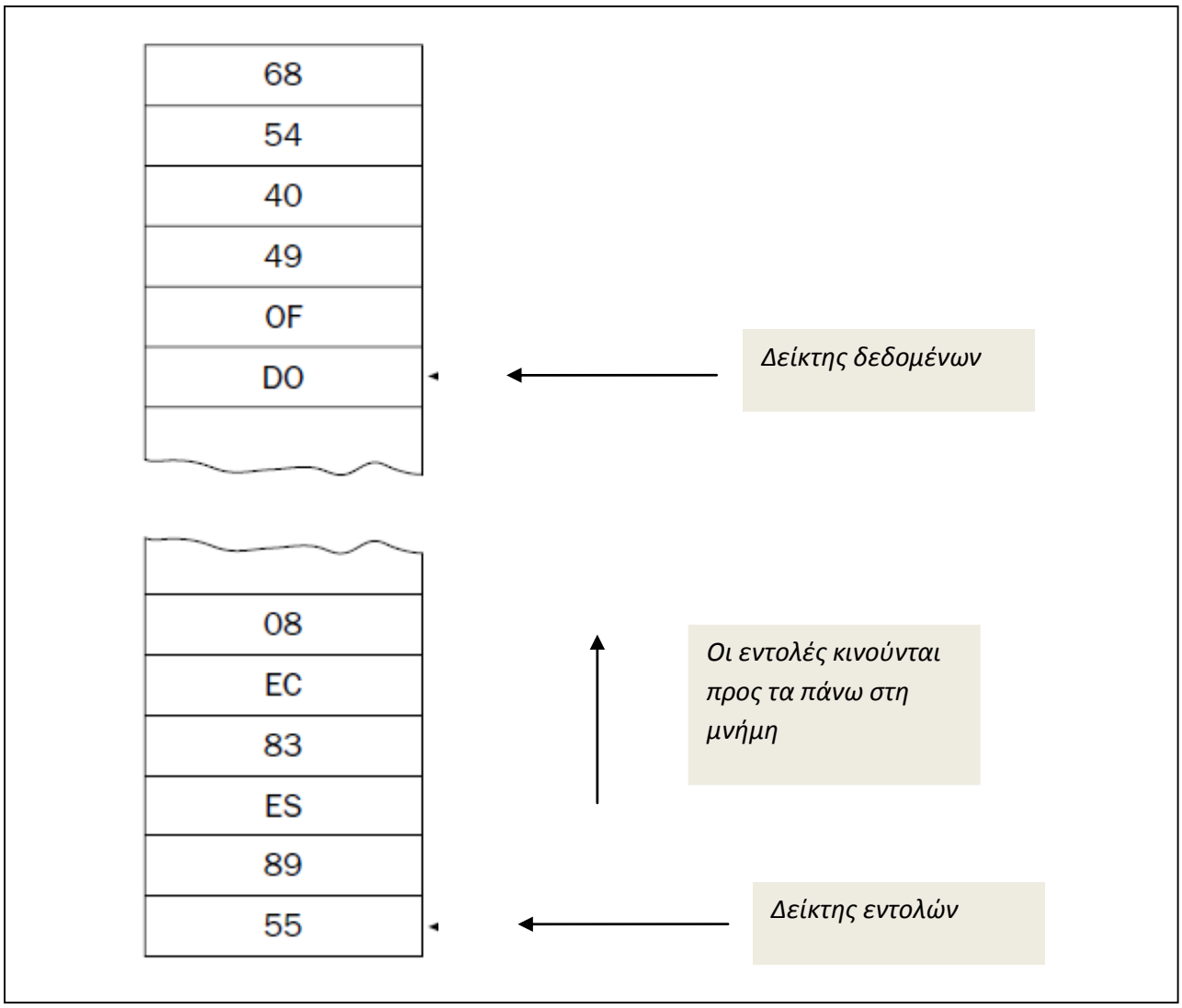
2.1 Το σύνολο εντολών του επεξεργαστή και οι συμβολική γλώσσα.

Οι εντολές στην συμβολική γλώσσα έχουν συνήθως μια αντιστοιχία ένα προς ένα με κάθε κώδικα λειτουργίας του επεξεργαστή με ελάχιστες εξαιρέσεις όπου συμβαίνει μια εντολή να συμβολίζει ένα σύνολο διαδοχικών λειτουργιών που θα εκτελεστούν από τον επεξεργαστή. Παρόλα αυτά όμως, στο μεγαλύτερο ποσοστό η παραπάνω δήλωση παρουσιάζεται ακριβής και αυτό είναι σημαντικό γιατί με αυτόν τον τρόπο δεν υπάρχουν ενδιάμεσα στάδια αφαίρεσης μεταξύ του κώδικα μηχανής και της γλώσσας του προγραμματιστή. Στην εποχή μας που υπάρχουν άπειρες γλώσσες προγραμματισμού και σε αυτές προστίθενται και εικονικές μηχανές πάνω στις οποίες εκτελούνται οι εντολές μιας γλώσσας υψηλού επιπέδου όπως είναι η Java και το .NET, είναι πολύ σημαντικό να γνωρίζουμε πώς όλα αυτά στο τέλος καταλήγουν στην συμβολική γλώσσα η οποία μας παρέχει μια σταθερή βάση.

Ο επεξεργαστής διαβάζει από τη μνήμη κώδικες εντολών (instruction codes) και δεδομένα. Κάθε κώδικας εντολής παρέχει στον επεξεργαστή πληροφορίες για την λειτουργία που πρέπει να εκτελεστεί. Το πρόβλημα που προκύπτει είναι πώς τα δεδομένα αποθηκεύονται στη μνήμη ακριβώς με τον ίδιο τρόπο που αποθηκεύονται οι εντολές και επομένως ο επεξεργαστής χρειάζεται έναν τρόπο για να τα διαχωρίζει και να ελέγχει τη ροή των εισόδων.

Η λύση στο πρόβλημα δίνεται από την ύπαρξη δύο ειδικών δεικτών (pointers), έναν που δείχνει το τμήμα που αποθηκεύονται τα δεδομένα στη μνήμη και έναν για τις αποθηκευμένες εντολές όπως φαίνεται και στην Εικόνα 2.1: Οι δείκτες στην μνήμη (Blum, 2005).

Έτσι λοιπόν ο δείκτης εντολών (instruction pointer) δείχνει τη θέση στη μνήμη στην οποία περιέχονται οι εντολές που θα εκτελεστούν. Αυτό βέβαια δεν απαγορεύει σε ορισμένες εντολές να αλλάξουν τη θέση του δείκτη ώστε να εκτελεστεί άλλο σύνολο εντολών. Αντίστοιχα ο δείκτης δεδομένων δείχνει στη μνήμη τη θέση στην οποία έχουν αποθηκευτεί τα δεδομένα. Τέλος, ένας κώδικας εντολής πρέπει να αποθηκεύεται με συγκεκριμένο τρόπο στην μνήμη ώστε να είναι αναγνώσιμος από τον επεξεργαστή και πρέπει να περιέχει τουλάχιστον έναν κώδικα λειτουργίας.



Εικόνα 2.1: Οι δείκτες στην μνήμη (Blum, 2005)

2.2 Επιλογή της κατάλληλης συμβολικής γλώσσας.

Κάθε επεξεργαστής, ανάλογα με την οικογένεια στην οποία ανήκει, έχει το δικό του ξεχωριστό σύνολο εντολών το οποίο αναγνωρίζει. Για παράδειγμα ένας επεξεργαστής της οικογένειας x86 έχει πολύ μικρότερο σύνολο εντολών από έναν της x64. Έτσι είναι εύλογα κατανοητό πως η συμβολική γλώσσα παρουσιάζει διάφορες παραλλαγές ανάλογα με την οικογένεια για την οποία γράφεται. Όταν λοιπόν κάποιος αναφέρεται στη συμβολική γλώσσα θα πρέπει να αναφέρει ακόμα για ποια αρχιτεκτονική επεξεργαστών συζητάει για να αποφεύγεται η σύγχυση. Ακόμα υπάρχουν διαφορές και στον τρόπο που οι

συμβολομεταφραστές μεταφράζουν την συμβολική γλώσσα με αποτέλεσμα να δημιουργείται ένα τεράστιο πλήθος επιλογών για τον προγραμματιστή το οποίο είναι ικανό να αποτελέσει το πρώτο σημαντικό εμπόδιο για αυτόν. Σαν αποτέλεσμα αφού πρώτα ο προγραμματιστής μελετήσει το περιβάλλον για το οποίο αναπτύσσεται η εφαρμογή, μπορεί να επιλέξει την κατάλληλη συμβολική γλώσσα, να την μελετήσει και στην συνέχεια να εφαρμόσει τη νεοαποκτηθείσα του γνώση. Το ίδιο όμως πράγμα ισχύει και για τον προγραμματιστή που χρησιμοποιεί την ΑΜΛ καθώς θα πρέπει πρώτα να μελετήσει το περιβάλλον για το οποίο αναπτύχθηκε η εφαρμογή και να μελετήσει την κατάλληλη συμβολική γλώσσα όπως θα δούμε και στη συνέχεια. Βέβαια υπάρχουν πολλά εργαλεία τα οποία αυτοματοποιούν πολύ μεγάλο μέρος από την απαιτούμενη δουλειά όμως η καλή γνώση της συμβολικής γλώσσας είναι ίσως το σημαντικότερο προαπαιτούμενο. Εξάλλου ουκ ολίγες φορές θα κληθεί ο προγραμματιστής να γράψει δικό του κώδικα ώστε να διαμορφώσει τη συμπεριφορά του προγράμματος-στόχου του.

2.3 Καταχωρητές.

Στην ΑΜΛ όμως όπως θα φανεί και στα επόμενα κεφάλαια είναι πολύ σημαντικό ο χρήστης-μηχανικός λογισμικού να κατανοήσει τη λειτουργία των καταχωρητών αλλιώς δε θα μπορεί να παρακολουθήσει τις μετακινήσεις των πληροφοριών καθώς και τις αλλαγές ροής στο πρόγραμμα. Γενικά οι τιμές στους καταχωρητές αποτελούν ίσως την πιο σημαντική πηγή πληροφοριών που θα χρειαστεί ο μηχανικός λογισμικού.

Τι ακριβώς είναι όμως οι καταχωρητές; Πρόκειται για πολύ γρήγορες μνήμες μικρής χωρητικότητας μέσα στον επεξεργαστή οι οποίες χρησιμοποιούνται για να αποθηκευτούν διευθύνσεις και δεδομένα μέχρι να χρησιμοποιηθούν από την κεντρική μονάδα επεξεργασίας. Στους επεξεργαστές της οικογένειας x86 της INTEL - η οποία αποτελεί και την οικογένεια επεξεργαστών με την οποία θα απασχοληθούμε στην συνέχεια της εργασίας – θα μπορούσαμε να ομαδοποιήσουμε τους καταχωρητές σε τέσσερις μεγάλες κατηγορίες: τους καταχωρητές γενικής χρήσης (general purpose registers), τους καταχωρητές τμημάτων (segment registers), τους καταχωρητές δεικτών (index/pointer registers) και τους καταχωρητές σημαιών (eflags registers).

2.3.1 Καταχωρητές γενικής χρήσης

Σε αυτήν την κατηγορία οι καταχωρητές είναι της τάξεως των 32bit αλλά μπορούν να χρησιμοποιηθούν και ως καταχωρητές των 8bit ή των 16bit. Εύκολα διαπιστώνεται και από το όνομα τους πως σε αυτούς τους καταχωρητές αποθηκεύονται συνεχώς πληροφορίες και εντολές και είναι οι πιο συχνά χρησιμοποιούμενοι από όλους τους καταχωρητές. Η κατάληξη «X» στο όνομα τους δόθηκε μεταγενέστερα από την λέξη «extended» που σημαίνει πως έχουν επεκταθεί στα 32bit. Το γράμμα «L» και το γράμμα «H» αντίστοιχα συμβολίζουν τα χαμηλά bit (low-end bit) και τα υψηλά bit (high-end bit) στο πρώτο τμήμα του καταχωρητή. Έτσι για παράδειγμα ο καταχωρητής EAX χωρίζεται ξεκινώντας από το πρώτο λιγότερο σημαντικό bit στον AL (0bit-7bit), τον AH (8bit-15bit) και τον AX (16bit-31bit) με σύνολο τα 32bit. Ο Πίνακας 2.1: Καταχωρητές Γενικής Χρήσης. απεικονίζει καθαρά τους καταχωρητές με τα ονόματα τους, το μέγεθος τους και τις πιο συνηθισμένες χρήσεις τους.

Πίνακας 2.1: Καταχωρητές Γενικής Χρήσης

Καταχωρητής	**Συνολικά Bit**	**31bit - 16bit**	**15bit - 8bit**	**7bit - 0bit**	**Ιδιότητα**
		Τμηματική ονομασία			
EAX (Extended Accumulator Register)	32	AX	AH	AL	Χρησιμοποιείται για είσοδο-έξοδο δεδομένων, αριθμητικές πράξεις και διακοπές
EBX (Extended Base Register)	32	BX	BH	BL	Χρησιμοποιείται σαν δείκτης βάσης για θέσεις μνήμης.

Καταχωρητής	Συνολικά Bit	31bit - 16bit	15bit - 8bit	7bit - 0bit	Ιδιότητα
ECX (Extended Count Register)	32	CX	CH	CL	Χρησιμοποιείται σαν μετρητής σε επαναλήψεις.
EDX (Extended Data Register)	32	DX	DH	DL	Χρησιμοποιείται για δεδομένα και συνήθως όπως ο EAX

2.3.2 Καταχωρητές τμημάτων

Σε αυτήν την κατηγορία οι καταχωρητές είναι της τάξεως των 16bit. Αποθηκεύουν πληροφορίες που αφορούν την διεύθυνση ενός συγκεκριμένου τμήματος της μνήμης. Η τιμής τους τροποποιείται μόνο από άλλους καταχωρητές γενικής χρήσης ή από ειδικές εντολές. Ο Πίνακας 2.2: Καταχωρητές Τμημάτων απεικονίζει καθαρά τους καταχωρητές με τα ονόματα τους, το μέγεθος τους και τις πιο συνηθισμένες χρήσεις τους.

Πίνακας 2.2: Καταχωρητές Τμημάτων

Καταχωρητής	Συνολικά Bit	Ιδιότητα
CS (Code Segment Register)	16	Αποθηκεύει τη θέση στην μνήμη που βρίσκεται ο κώδικας του εκτελέσιμου
DS (Data Segment Register)	16	Αποθηκεύει τη θέση στην μνήμη που βρίσκονται τα δεδομένα του εκτελέσιμου τα οποία θα ανακτηθούν κατά την εκτέλεση

Καταχωρητής	Συνολικά Bit	Ιδιότητα
ES, FS, GS	16	Δείκτες οι οποίοι αναφέρονται σε μακρινές περιοχές της μνήμης όπως η μνήμη της κάρτας γραφικών.
SS (Stack Segment Register)	16	Αποθηκεύει τη διεύθυνση του σωρού που χρησιμοποιεί το εκτελέσιμο.

2.3.3 Καταχωρητές δείκτες

Οι καταχωρητές δείκτες όπως φαίνεται και από το όνομα τους χρησιμοποιούνται για τον έμμεσο υπολογισμό διευθύνσεων κυρίως αλλά και για αριθμητικές ή λογικές πράξεις και περιστροφές. Έχουν μέγεθος 32bit αλλά μπορεί να χρησιμοποιηθεί μόνο το χαμηλό μέρος τους με 16bit εκτός αν βρίσκονται σε κατάσταση προστασίας. Ο καθένας συνήθως χρησιμοποιείται για διαφορετικούς σκοπούς. Ο

Πίνακας 2.3: Καταχωρητές Δείκτες απεικονίζει καθαρά τους καταχωρητές με τα ονόματα τους, το μέγεθος τους και τις πιο συνηθισμένες χρήσεις τους στο τμήμα στο οποίο χρησιμοποιούνται.

Πίνακας 2.3: Καταχωρητές Δείκτες

Καταχωρητής	Συνολικά Bit	31bit – 16bit	15bit – 0bit	Ιδιότητα
			Τμηματική ονομασία	
DS: ESI (Extended Source Index Register)	32	-	SI	Δείκτης προέλευσης (Source Index)

Καταχωρητής	Συνολικά Bit	31bit – 16bit	15bit – 0bit	Ιδιότητα
			Τμηματική ονομασία	
ES: EDI (Extended Destination Index)	32	-	DI	Δείκτης προορισμού (Destination Index)
SS: ESP (Extended Stack Pointer)	32	-	SP	Δείκτης σωρού (Stack Pointer)
SS: EBP (Extended Base Pointer)	32	-	BP	Δείκτης βάσης (Base Pointer)
CS: EIP (Extended Instruction Pointer)	32	-	IP	Δείκτης της επόμενης εντολής.

2.3.4 Καταχωρητές σημαιών ή κατάστασης (EFLAGS)

Οι καταχωρητές σημαιών συνήθως έχουν μέγεθος 16 μεμονωμένων bit και χρησιμοποιούνται για να κρατήσουν την κατάσταση κάποιας προηγούμενης εντολής. Τροποποιούνται μόνο εξαιτίας κάποιων εντολών και επίσης σαν σημαίες για ελέγχους. . Στα επόμενα κεφάλαια θα παρουσιαστεί η χρησιμότητα τους όταν θα θελήσουμε να αλλάξουμε τη ροή του προγράμματος επιβάλλοντας μεταβολές στις τιμές τους για να φανούν κάποιες αδυναμίες στις επιθέσεις στον κώδικα. Ο Πίνακας 2.4: Καταχωρητές Σημαιών (Toronto University, 2010) απεικονίζει καθαρά τους καταχωρητές με τα ονόματα τους, το μέγεθος τους και τις πιο συνηθισμένες χρήσεις τους στο τμήμα στο οποίο χρησιμοποιούνται.

Πίνακας 2.4: Καταχωρητές Σημαιών (Toronto University, 2010)

Συνολικά Bit	**Καταχωρητής**	**Ιδιότητα**
0	CF	Carry flag
2	PF	Parity flag
4	AF	Auxiliary carry flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O Priviledge level
14	NT	Nested task Flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check flag (486+)
19	VIF	Virutal interrupt flag
20	VIP	Virtual interrupt pending flag
21	ID	ID flag

2.4 Σύνοψη

Ανακεφαλαιώνοντας την ενότητα αυτή σίγουρα παρατηρεί κανείς την σημασία που έχει η συμβολική γλώσσα για τον μηχανικό λογισμικού ή τον απλό προγραμματιστή καθώς βρίσκεται ένα στάδιο παραπάνω από τη γλώσσα μηχανής χωρίς να παρεμβάλλεται κάποιο ενδιάμεσο στάδιο αφαίρεσης με αποτέλεσμα να αποκτά καλύτερη άποψη για τη συμπεριφορά του υπολογιστή και να μπορεί να προβλέψει δυσμενείς για αυτόν καταστάσεις. Αν ο αναγνώστης έχει αποκτήσει έναν μεγαλύτερο βαθμό οικειότητας με όρους όπως κώδικας λειτουργίας (opcode), κώδικας εντολής (instruction code), μνημονικά σίγουρα η συμβολική γλώσσα δε θα μοιάζει τόσο ξένη και δυσπρόσιτη όσο αρχικά φαινόταν.

Μπορεί όμως εύκολα να αναρωτηθεί κανείς σε τι χρησιμεύει η Assembly και τι ρόλο επιτελεί στην αντίστροφη μηχανική λογισμικού. Για να απαντήσει κανείς σε αυτό θα πρέπει να περιγράψει πρώτα την ΑΜΛ σαν τον τομέα που σκοπεύει να μετατρέψει τον μη αναγνώσιμο κώδικα μηχανής σε ευανάγνωστο πηγαίο κώδικα μιας γλώσσας υψηλού επιπέδου. Όταν κάποιος κατανοήσει τι πρέπει να γίνει βλέπει άμεσα πως πριν μετατραπεί ο κώδικας μηχανής στην τελική του μορφή, αναπόφευκτα θα πρέπει να περάσει από το ενδιάμεσο στάδιο της μετατροπής του στην συμβολική γλώσσα. Επομένως η συμβολική γλώσσα αποτελεί τον ακρογωνιαίο λίθο της ΑΜΛ, και χωρίς αυτήν τη βάση δεν μπορεί να γίνει καμία πρόοδος.

Καθώς η εργασία είναι προσανατολισμένη στην ΑΜΛ ενός φορητού εκτελέσιμου αρχείου των Windows (δηλαδή τη μορφή αρχείου που χρησιμοποιείται για προγράμματα και εκτελέσιμα αρχεία που συνδέονται για να σχηματίσουν εκτελέσιμα προγράμματα στα Windows), στο επόμενο κεφάλαιο θα γίνει μια προσπάθεια ανάλυσης της δομής του φορητού εκτελέσιμου και του τρόπου με τον οποίο το διαχειρίζεται ο υπολογιστής. Ο στόχος είναι η πρώτη πραγματική επαφή με τον κώδικα μηχανής όπως αυτός έχει αποθηκευθεί στο εκτελέσιμο και η εξοικείωση με αυτό.

3. Το φορητό εκτελέσιμο αρχείο των windows (Win32 Portable Executable)

Εισαγωγή

Στην ΑΜΛ όπως αναφέρθηκε στα προηγούμενα κεφάλαια η βασική πηγή πληροφοριών και πολλές φορές η μοναδική πηγή είναι το εκτελέσιμο αρχείο της εφαρμογής και για αυτόν τον λόγο σε αυτό το κεφάλαιο θα γίνει μια προσπάθεια να αναλυθούν η δομή του ΦΕ - φορητού εκτελέσιμου - αρχείου των windows (Win32 Portable Executable). Μέσα από σχεδιαγράμματα θα παρουσιαστούν οι διαφορετικοί τομείς που απαρτίζουν το ΦΕ καθώς επίσης και πως αποθηκεύονται οι πληροφορίες που αφορούν την σύνδεση του ΦΕ με δυναμικές βιβλιοθήκες (Dynamic Link Library, DLL).

Στόχος είναι η καλύτερη κατανόηση της σωστής διαδικασίας φόρτωσης ενός ΦΕ στην μνήμη από το λειτουργικό ώστε να λειτουργήσει η εφαρμογή. Σε αυτό το σημείο απλά να αναφέρουμε πως μία από τις τεχνικές άμυνας ενός ΦΕ από Crackers που θέλουν να «σπάσουν» το προστατευμένο αρχείο αποτελεί η αλλοίωση της δομής του ΦΕ με τρόπο τέτοιο ώστε να είναι αδύνατον να φορτωθεί σωστά σε διάφορα εργαλεία που χρησιμοποιούνται, δυσχεραίνοντας έτσι την ανάλυση του, χωρίς να επηρεάζει την κανονική του λειτουργία από το λειτουργικό σύστημα.

Έτσι λοιπόν είναι ευνόητο πως η κατανόηση της λειτουργίας ενός ΦΕ είναι εξίσου σημαντική με την συμβολική γλώσσα για κάποιον που θέλει να εμβαθύνει στην επιστήμη της ΑΜΛ. Ακόμα όμως και αν εξαιρέσει κανείς την επίπτωση στην ΑΜΛ, η γνώση της λειτουργίας του ΦΕ μπορεί να βοηθήσει τον προγραμματιστή στη βελτίωση της ποιότητας του. Ακόμα μπορεί να βοηθήσει και στον προγραμματισμό επαναχρησιμοποιήσιμων αρχείων DLL. Θα μπορούσε να πει κανείς ότι βοηθάει στην βελτίωση της ποιότητας των εφαρμογών και των ικανοτήτων του προγραμματιστή.

3.1 Το φορητό εκτελέσιμο.

Το ΦΕ όπως το γνωρίζουμε σήμερα έχει υποστεί πολλές τροποποιήσεις από την αρχική του μορφή. Για ιστορικούς και μόνο λόγους να αναφέρουμε πως αποτελεί μια παραλλαγή του προτύπου COFF (Common Object File Format) που εμφανίστηκε πρώτη φορά στα συστήματα Unix. Για αυτόν το λόγο ακόμα και σήμερα πολλές φορές ονομάζεται και PE-COFF δηλαδή Portable Executable And Common Object File Format που ίσως αποτελεί και την πιο σωστή του ονομασία. Προς αποφυγήν σύγχυσης σε όλη την έκταση της εργασίας με τον όρο ΦΕ θα αναφερόμαστε πάντα στο προαναφερθέν πρότυπο.

Η ανάγκη που οδήγησε στην δημιουργία του ΦΕ ήταν η ύπαρξη ενός ενιαίου προτύπου για κάθε λειτουργικό σύστημα Windows και κάθε μετεξέλιξη αυτών, το οποίο θα διατηρούσε την προς τα πίσω συμβατότητα. Από το 1995 και μετά τα Windows 95 αυτό σε μεγάλο βαθμό επιτεύχθηκε. Ένα σημαντικό στοιχείο είναι ότι το ΦΕ διατηρεί την κεφαλή MZ (MZ Header) για να υπάρχει συμβατότητα με MS-DOS εφαρμογές. Στην συνέχεια μέχρι και σήμερα, ακόμα και με την εμφάνιση επεξεργαστών 64bit δεν παρουσιάστηκαν ιδιαίτερες αλλαγές στο πρότυπο του ΦΕ. Απλά κάποια πεδία αχρηστεύτηκαν και αφαιρέθηκαν και δεν προστέθηκε κάτι. Το νέο πρότυπο είναι γνωστό και ως PE32+.

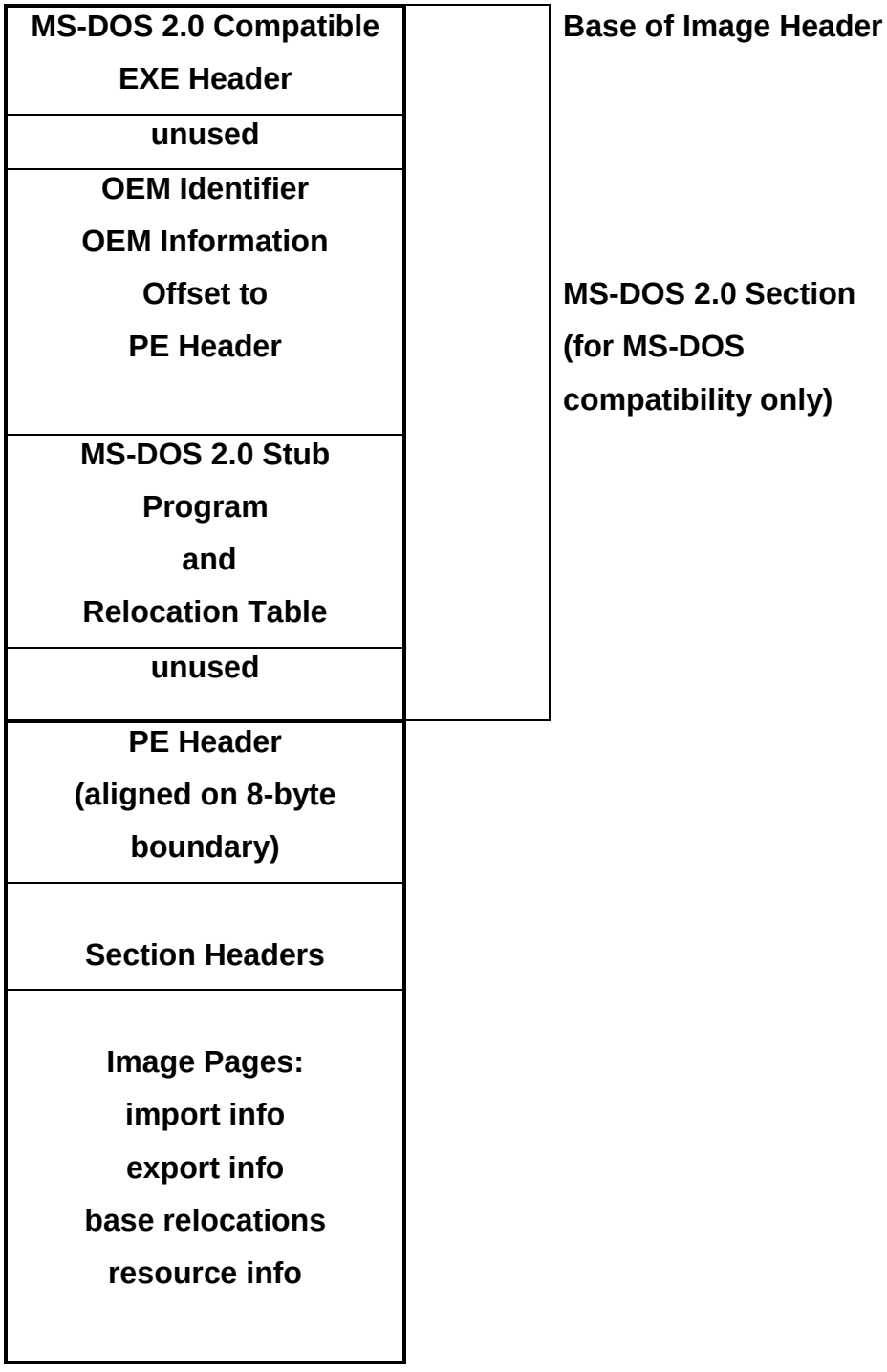
Τέλος, ένα ενδιαφέρον στοιχείο προκύπτει από τη σύγκριση ενός ΦΕ και μιας δυναμικής βιβλιοθήκης δηλαδή ενός αρχείου DLL. Και τα δύο αρχεία αν τα παρατηρήσει κάποιος έχουν ακριβώς την ίδια δομή με μόνη διαφορά ενός Bit. Το ένα αυτό Bit ορίζει στο λειτουργικό σύστημα πως θα διαχειριστεί το αρχείο.

3.2 Η δομή του φορητού εκτελέσιμου.

Προχωρώντας στην ανάλυση ενός ΦΕ, αρχικά πρέπει να καταλάβει κανείς τη δομή του. Ένα ΦΕ είναι χωρισμένο σε τομείς οι οποίοι χωρίζουν λογικά τα δεδομένα που περιέχουν. Δηλαδή κάθε τομέας αποτελεί μια ομάδα δεδομένων που επιτελούν κάποιον σκοπό. Επίσης κάθε τομέας έχει και το δικό του αντιπροσωπευτικό όνομα που δίνει πληροφορίες για την λειτουργία του.

Η

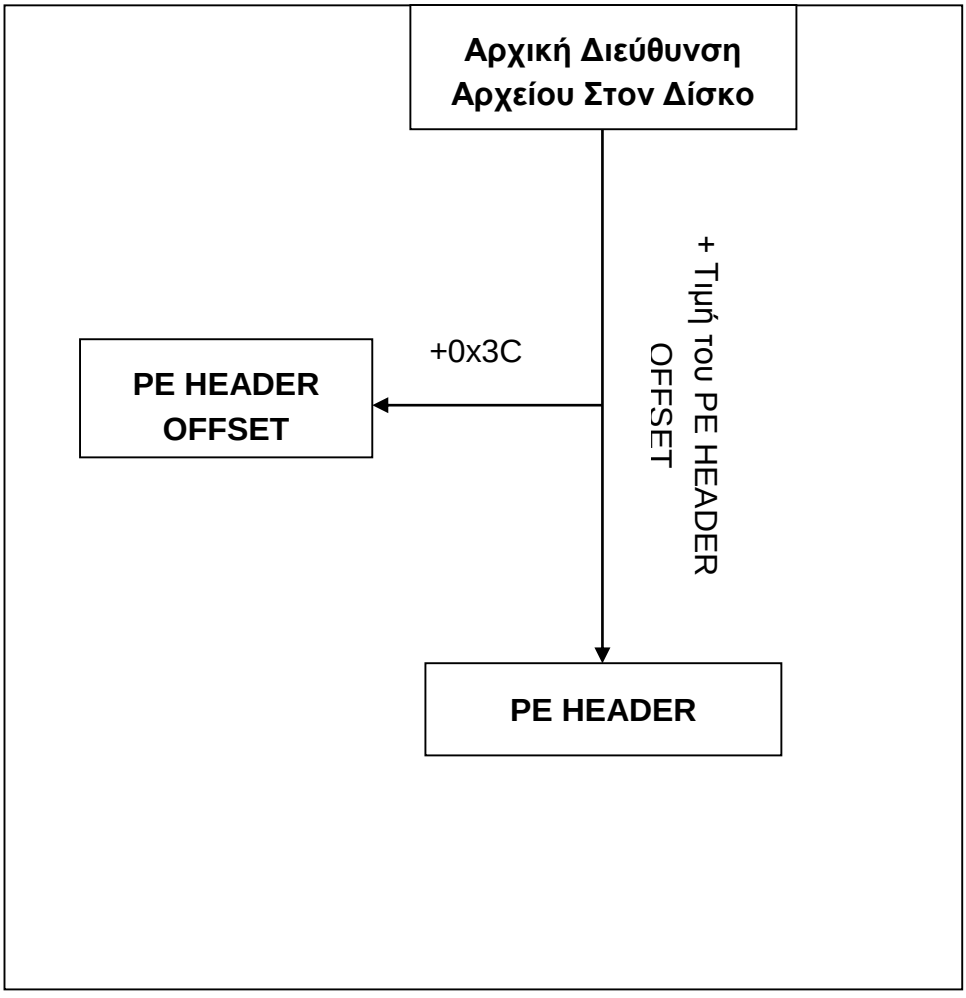
Εικόνα 3.1: Η δομή ενός τυπικού ΦΕ (Microsoft, 2006) απεικονίζει ξεκάθαρα τους τομείς στους οποίους χωρίζεται το ΦΕ με τις ονομασίες τους.



Εικόνα 3.1: Η δομή ενός τυπικού ΦΕ (Microsoft, 2006)

3.2.1 Τομέας MS-DOS

Στην αρχή παρατηρούμε όπως προαναφέραμε ότι υπάρχει ένας τομέας αφιερωμένος σε πληροφορίες που υπάρχουν για λόγους συμβατότητας με το MS-DOS καθώς και ένα κομμάτι κώδικα, ένα πρόγραμμα το οποίο είναι προγραμματισμένο να «τρέξει» όταν το ΦΕ εκτελεστεί σε περιβάλλον MS-DOS. Στην ΑΜΛ το πιο σημαντικό στοιχείο που απασχολεί τον μηχανικό λογισμικού και βρίσκεται σε αυτόν τον τομέα είναι η ανεύρεση της τιμής μετατόπισης (offset) από την αρχική διεύθυνση του αρχείου στον δίσκο που θα οδηγήσει στην διεύθυνση της κεφαλής του ΦΕ (PE Header). Συνήθως αυτή η τιμή βρίσκεται στο offset 0x3C (οι τιμές διευθύνσεων και offset δίνονται πάντα στο 16δικό σύστημα) από την αρχή του αρχείου. Η τιμή αυτή έχει μεγάλη σημασία και για το λειτουργικό σύστημα γιατί του επιτρέπει να αναγνωρίσει το αρχείο και να το εκτελέσει παρ' ότι υπάρχει το κομμάτι του MS-DOS Stub προγράμματος. Μια απεικόνιση της διαδικασίας αυτής φαίνεται καλύτερα στην Εικόνα 3.2: Εύρεση της κεφαλής PE.



Εικόνα 3.2: Εύρεση της κεφαλής PE

Επομένως ο τομέας που περιέχει τις MS-DOS πληροφορίες στην ουσία δεν απασχολεί καθόλου τον μηχανικό λογισμικού (reverser). Αντίθετα οι περισσότερες σημαντικές πληροφορίες που θα τον απασχολήσουν βρίσκονται στο κομμάτι του PE Header.

3.2.2 Η κεφαλή PE (PE Header)

Έχοντας εντοπίσει την αρχή του PE Header βλέπει κανείς αρχικά ένα πεδίο αποτελούμενο από 4 byte που αποτελεί την υπογραφή (signature) η οποία χαρακτηρίζει το αρχείο ως ΦΕ. Ειδικότερα τα 4 byte αυτά έχουν την τιμή PE\0\0 όπου το \0 είναι δυαδικό 0. Είναι σημαντικό να τονίσουμε πως η υπογραφή υπάρχει μόνο στα εκτελέσιμα (executable ή αλλιώς γνωστά και σαν Image) αρχεία και όχι στα Object αρχεία.

3.2.3 Η κεφαλή COFF (COFF File Header)

Στην συνέχεια ακολουθεί η κεφαλή COFF η οποία υπάρχει και σε Image και σε Object αρχεία. Σε αυτό το κομμάτι βρίσκονται πληροφορίες σχετικά με το είδος της μηχανής για την οποία έχει δημιουργηθεί το αρχείο, για παράδειγμα στα αρχικά δύο byte θα έχει διαφορετική τιμή για επεξεργαστές AMD64 ή IA32. Πρόκειται για μια πολύ σημαντική πληροφορία γιατί σε διαφορετικές αρχιτεκτονικές μπορεί να υπάρχουν σημαντικές αλλαγές στον τρόπο που φορτώνεται το εκτελέσιμο στην μνήμη και άλλες διαφορές (π.χ. αν χρησιμοποιεί Little ή Big Endian) που μπορεί να οδηγήσουν τον μηχανικό λογισμικού σε αδιέξοδα. Ακόμα ο μηχανικός λογισμικού μπορεί να αντλήσει πληροφορίες που αφορούν την ημερομηνία που δημιουργήθηκε το αρχείο, το πλήθος των τομέων που το απαρτίζουν καθώς και το μέγεθος του προαιρετικού τμήματος κεφαλής που υπάρχει σε αρχεία Image αλλά όχι σε Object. Στοιχεία που σχεδόν ποτέ δε θα υπάρχουν για να μη τον διευκολύνουν αποτελούν τα σύμβολα αποσφαλμάτωσης (debugging symbols). Εξάλλου από το πρότυπο δεν προτείνεται να υπάρχουν όμως στην περίπτωση που υπάρχουν θα βρίσκονται σίγουρα σε αυτό το τμήμα. Τέλος περιέχει πληροφορίες που αφορούν ιδιαίτερα χαρακτηριστικά του αρχείου όπως για παράδειγμα αν είναι εκτελέσιμο ή

δυναμική βιβλιοθήκη. Σίγουρα στην προσπάθεια του κανείς να αντλήσει όσο το δυνατόν περισσότερες πληροφορίες για τον στόχο του θα πρέπει να ανατρέχει συνέχεια στην τεκμηρίωση που παρέχει η Microsoft για τη δομή του ΦΕ.

3.2.4 Η προαιρετική κεφαλή (Optional Header)

Προχωρώντας στην ανάλυση της δομής του ΦΕ βλέπει κανείς ένα τμήμα που περιέχει μια προαιρετική κεφαλή (Optional Header). Η δουλειά αυτού του τμήματος είναι να παρέχει περαιτέρω πληροφορίες στον φορτωτή (loader). Ο όρος «προαιρετική» είναι λίγο καταχρηστικός στην συγκεκριμένη περίπτωση βέβαια γιατί ένα εκτελέσιμο αρχείο (Image) θα πρέπει οπωσδήποτε να έχει αυτό το τμήμα σε αντίθεση με τα Object αρχεία στα οποία δε διαδραματίζει κανένα ρόλο ακόμα και όταν υπάρχει.

Στην ΑΜΛ αυτός ο τομέας παρέχει πολύ ενδιαφέρουσες πληροφορίες όπως το αρχικό μέγεθος του σωρού. Ακόμα μπορούν να αντληθούν πληροφορίες για την προτιμώμενη αρχική διεύθυνση στη μνήμη και για την έκδοση του λειτουργικού συστήματος. Επίσης εδώ φαίνεται το σημείο εισόδου (entry point) του εκτελέσιμου αρχείου που αποτελεί το σημείο από το οποίο θα αρχίσει η περαιτέρω έρευνα για την ροή εκτέλεσης του. Η σημασία του θα παρουσιαστεί καλύτερα σε παρακάτω κεφάλαια όταν θα παρουσιάσουμε τις τεχνικές που χρησιμοποιούν πολλές μέθοδοι προστασίας για να το αποκρύψουν πριν το πρόγραμμα φορτωθεί στην μνήμη ενώ επαναφέρουν τη ροή εκτέλεσης σε αυτό μόνο εφόσον έχει ολοκληρωθεί η επιτυχής εκτέλεση του δικού τους κώδικα.

Τα πεδία της προαιρετικής κεφαλής μπορούν να κατηγοριοποιηθούν σε τρεις ομάδες, τα τυπικά πεδία (Standard Fields), τα ειδικά πεδία των Windows (Windows Specific Fields) και τους καταλόγους δεδομένων.

Τυπικά πεδία

Σε αυτά τα πεδία υπάρχουν γενικές πληροφορίες που αφορούν την σύνδεση και την φόρτωση του εκτελέσιμου. Τα πιο σημαντικά πεδία με την σημασία τους για την ΑΜΛ παρουσιάζει ο Πίνακας 3.1: Τα τυπικά πεδία της προαιρετικής κεφαλής.

Πίνακας 3.1: Τα τυπικά πεδία της προαιρετικής κεφαλής

Όνομα πεδίου	**Σημασία**
Magic	Αποτελεί ένα είδος υπογραφής από το οποίο καταλαβαίνει κανείς το είδος του εκτελέσιμου αρχείου.
SizeOfCode	Υποδεικνύει το άθροισμα των μεγεθών των τομέων που περιέχουν τον κώδικα. Κατά την προσθήκη κώδικα στο εκτελέσιμο θα πρέπει να ενημερωθεί το πεδίο.
size of initialized data	Υποδεικνύει το άθροισμα των μεγεθών των τομέων που περιέχουν αρχικοποιημένα δεδομένα. Χρησιμοποιείται ομοίως με το προηγούμενο.
size of uninitialized data	Υποδεικνύει το άθροισμα των μεγεθών των τομέων που περιέχουν μη αρχικοποιημένα δεδομένα. Χρησιμοποιείται ομοίως με το προηγούμενο.
AddressOfEntryPoint	Περιέχει την τιμή μετατόπισης (offset) με την οποία θα βρεθεί η διεύθυνση από την οποία αρχίζει η εκτέλεση του κώδικα του αρχείου. Είναι μια σχετική εικονική διεύθυνση (RVA), δηλαδή μια τιμή μετατόπισης από την διεύθυνση στην οποία έχει φορτωθεί το αρχείο στην μνήμη. Πολλές φορές ίσως χρειαστεί να τροποποιηθεί το σημείο εισόδου για να αποφευχθεί ο κώδικας προστασίας του εκτελέσιμου ή θελήσει ο μηχανικός λογισμικού να αλλάξει την συμπεριφορά του εκτελέσιμου. Ακόμα, κακόβουλο λογισμικό μπορεί να επιδείξει παρόμοια συμπεριφορά, τροποποιώντας το σημείο εισόδου
Όνομα πεδίου	**Σημασία**
AddressOfEntryPoint (συνέχεια)	ώστε να εκτελέσει το δικό του τμήμα κώδικα πριν την κανονική εκτέλεση του προγράμματος.

BaseOfCode	Πρόκειται για άλλη μια RVA που υποδεικνύει την αρχική διεύθυνση του τμήματος που περιέχει τον εκτελέσιμο κώδικα όταν φορτωθεί στην μνήμη.
BaseOfData (δεν υπάρχει σε PE32+ αρχεία)	Ομοίως με το BaseOfCode αλλά για δεδομένα.

Ειδικά πεδία των Windows

Σε αυτά τα πεδία υπάρχουν πληροφορίες που χρησιμεύουν κατά την σύνδεση και την φόρτωση του εκτελέσιμου ειδικά στα Windows. Τα πιο σημαντικά πεδία με την σημασία τους για την ΑΜΛ παρουσιάζει ο Πίνακας 3.2: Τα ειδικά πεδία των Windows στην προαιρετική κεφαλή.

Πίνακας 3.2: Τα ειδικά πεδία των Windows στην προαιρετική κεφαλή

Όνομα πεδίου	**Σημασία**
ImageBase	Αποτελεί την προτιμώμενη διεύθυνση της εικόνας του εκτελέσιμου όταν φορτωθεί στην μνήμη. Όπως φανερώνει η λέξη «προτιμώμενη» ο φορτωτής έχει την δυνατότητα να προσπεράσει αυτό το πεδίο και να χρησιμοποιήσει άλλη διεύθυνση. Πολλές φορές στον υπολογισμό διευθύνσεων οι υπολογισμοί θα πρέπει να περιλαμβάνουν και την τιμή αυτού του πεδίου.
SizeOfImage	Το μέγεθος της εικόνας του εκτελέσιμου (σε bytes) όπως αυτό φορτώνεται στην μνήμη. Όταν τροποποιηθεί το εκτελέσιμο πρέπει να αλλάξει και η τιμή του πεδίου αυτού.
SizeOfHeaders	Όπως και παραπάνω αλλά για τις κεφαλές του εκτελέσιμου.

Όνομα πεδίου	Σημασία
CheckSum	Σε αυτό το πεδίο αποθηκεύεται το άθροισμα ελέγχου (checksum) του αρχείου εικόνας. Ο αλγόριθμος ελέγχου είναι ενσωματωμένος στην δυναμική βιβλιοθήκη «imagehelp.dll» και ο έλεγχος του γίνεται για τα εκτελέσιμα αρχεία και τις δυναμικές βιβλιοθήκες κατά την φόρτωση. Με την τροποποίηση του εκτελέσιμου ευνόητο είναι πως πρέπει να αλλάξει και η τιμή αυτή.

Τέλος υπάρχουν και κάποια πεδία που αφορούν χαρακτηριστικά DLL και υποσυστήματα των windows.

Για την ΑΜΛ τα μόνα πεδία που παρουσιάζουν πραγματικό ενδιαφέρον είναι κάποια χαρακτηριστικά για αρχεία DLL όπως:

1. Το πεδίο IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE με τιμή 0x0040 το οποίο σημαίνει πως το αρχείο DLL μπορεί να μεταφερθεί σε άλλη θέση στην μνήμη κατά την φόρτωση του.
2. Το πεδίο IMAGE_DLL_CHARACTERISTICS_FORCE_INTEGRITY με τιμή 0x0080 η ύπαρξη του οποίου σημαίνει πως είναι υποχρεωτικοί οι έλεγχοι ακεραιότητας του κώδικα.
3. Το πεδίο IMAGE_DLL_CHARACTERISTICS_NX_COMPAT από τα αρχικά No EXECUTE και με τιμή 0x0100 το οποίο υποδεικνύει την συμβατότητα του αρχείου με την εκάστοτε τεχνολογία μη εκτέλεσης κώδικα που βρίσκεται σε χώρο της μνήμης που έχει δεσμευθεί για αποθήκευση δεδομένων. Στην Microsoft η συγκεκριμένη τεχνολογία ονομάζεται Data Execution Prevention (DEP), η AMD την ονομάζει Execute Disable (XD) ενώ επίσης παρουσιάζεται και με πολλές διαφορετικές ονομασίες. Πραγματικά ενδιαφέρον είναι πως με την αύξηση των επιθέσεων υπερχείλισης της προσωρινής μνήμης (buffer overflow attack) από κακόβουλο λογισμικό, απέκτησε ξεχωριστή σημασία η δυνατότητα μη-εκτέλεσης

κώδικα στα σύγχρονα συστήματα με αποτέλεσμα να παρουσιάζεται αυτή η μέθοδος προστασίας σελίδων σε όλες τις αρχιτεκτονικές.

4. Το πεδίο IMAGE_DLLCHARACTERISTICS_NO_SEH από τα αρχικά No Structured Exception Handling και με τιμή 0x0400 αποτρέποντας την ύπαρξη κάποιου δείκτης προς το DLL αρχείο σε περίπτωση που παρουσιαστεί κάποια εξαίρεση.

Κατάλογοι Δεδομένων

Ιδιαίτερη σημασία παρουσιάζουν στην ΑΜΛ οι κατάλογοι δεδομένων καθώς για παράδειγμα η μερική καταστροφή τους από λογισμικό προστασίας το οποίο ύστερα αναλαμβάνει την επαναδημιουργία τους, αποτελεί μια από τις συνηθέστερες ίσως πρακτικές που ακολουθούνται σε εμπορικές εφαρμογές. Με αυτόν τον τρόπο αν κάποιος παρακάμψει ή αφαιρέσει, όπως θα αναλυθεί στο Κεφάλαιο 6, τον κώδικα του λογισμικού προστασίας στο τέλος το μόνο που του απομένει είναι ένα εκτελέσιμο αρχείο το οποίο δεν θα έχει τα κατάλληλα δεδομένα ώστε να φορτωθεί σωστά στην μνήμη και να εκτελεστεί.

Οι κατάλογοι δεδομένων στην προαιρετική κεφαλή αποτελούν έναν πίνακα με 16 αντικείμενα τα οποία με τη σειρά τους αποτελούνται από τιμές μεγέθους 8 byte. Από τα 16 αντικείμενα, ο πίνακας εξαγωγής (export table) γνωστός και ως .edata τομέας και ο πίνακας εισαγωγής (import table) γνωστός και ως .idata τομέας είναι αυτοί που έχουν την μεγαλύτερη σημασία στην ΑΜΛ.

<u>.edata</u>

Ο πίνακας αυτός περιέχει πληροφορίες που αφορούν σύμβολα τα οποία εξάγονται μέσα από δυναμικές συνδέσεις. Οι πληροφορίες αυτές συνήθως είναι διευθύνσεις προς εξαγόμενες συναρτήσεις (functions), δηλαδή υπορουτίνες οι οποίες υπάρχουν ήδη σε κάποια δυναμική βιβλιοθήκη και εφόσον γίνει η σύνδεση του αρχείου μπορούν απλά να κληθούν με το όνομα τους. Όπως είναι εύκολα κατανοητό αυτός ο πίνακας αφορά κυρίως DLL αρχεία γιατί συνήθως αυτά αποτελούνται από συναρτήσεις και σύμβολα τα οποία χρησιμοποιούνται από άλλα αρχεία, δηλαδή εξάγονται σε αυτά. Πολλές φορές όμως είναι δυνατόν ένα αρχείο DLL να χρησιμοποιεί και εισαγόμενα σύμβολα ή συναρτήσεις από άλλα αρχεία DLL τα οποία θα πρέπει να αναφέρονται με την σειρά τους στον τομέα .**idata.**

.idata:

Directory Table
Null Directory Entry
DLL1 Import Lookup Table
Null
DLL2 Import Lookup Table
Null
DLL3 Import Lookup Table
Null
Hint-Name Table

Εικόνα 3.3: Δομή ενός τυπικού τομέα i.data (Microsoft, 2006)

Ο τομέας αυτός υπάρχει σε οποιοδήποτε εκτελέσιμο αρχείο ή αρχείο DLL χρησιμοποιεί εισαγόμενες συναρτήσεις ή σύμβολα. Ουσιαστικά αυτό σημαίνει ότι υπάρχει **σχεδόν σε όλα** τα ΦΕ. Ακόμα, τις περισσότερες φορές στην ΑΜΛ ο μηχανικός λογισμικού θα πρέπει να επιδιορθώσει τον τομέα αυτόν αφού παρακάμψει την προστασία του ΦΕ επομένως είναι απαραίτητο να γνωρίζει τη δομή του και τις ιδιαιτερότητες του. Στην Εικόνα 3.3: Δομή ενός τυπικού τομέα i.data (Microsoft, 2006) φαίνεται η δομή του τομέα i.data σε ένα τυπικό ΦΕ.

Έτσι λοιπόν στην αρχή του τομέα των πληροφοριών για εισαγωγή, υπάρχει ένας πίνακας καταλόγου εισαγωγής (Import Directory Table) ο οποίος παρέχει ουσιαστικά μια περιγραφή των πληροφοριών που θα ακολουθήσουν. Κάθε καταχώρηση στον πίνακα

περιλαμβάνει πληροφορίες που βοηθούν στην ανεύρεση της διεύθυνσης του σημείου εισόδου ενός DLL. Επομένως ο πίνακας αυτός είναι ουσιαστικά ένας πίνακας καταχωρήσεων αρχείων DLL. Τα κυριότερα πεδία που παρουσιάζουν ενδιαφέρον στην ΑΜΛ παρουσιάζει ο Πίνακας 3.3: Δομή της εγγραφής του πίνακα καταλόγου εισαγωγής.

Πίνακας 3.3: Δομή της εγγραφής του πίνακα καταλόγου εισαγωγής

Offset	**Πεδίο**	**Σημασία**
0	Import Lookup Table (ILT) RVA	Πρόκειται για την σχετική διεύθυνση που δείχνει την αρχή του πίνακα αναζήτησης εισαγωγών (Import Lookup Table). Ο πίνακας αυτός με τη σειρά του έχει μια καταχώρηση για κάθε εισαγωγή.
12	Name RVA	Πρόκειται για την σχετική διεύθυνση που δείχνει την συμβολοσειρά (String) που δηλώνει το όνομα ενός DLL αρχείου.
16	Import Address Table (IAT) RVA	Πρόκειται για την σχετική διεύθυνση που δείχνει την αρχή του πίνακα διευθύνσεων εισαγωγών (Import Address Table). Ο πίνακας αυτός έχει την ίδια δομή στις εγγραφές του όπως και ο ILT.

3.3 Σύνοψη

Σε αυτήν την ενότητα, μετά από μια γενική αναφορά στην ιστορία και την μετέπειτα εξέλιξη του προτύπου PE-COFF (Portable Executable - Common Object File Format), σίγουρα μπορεί κάποιος να κατανοήσει καλύτερα πόσες σημαντικές πληροφορίες μπορεί κανείς να συγκεντρώσει ρίχνοντας μια γρήγορη ματιά σε ένα εκτελέσιμο αρχείο. Ακόμα κατανοώντας καλύτερα την δομή του ΦΕ (φορητού εκτελέσιμου) αποκτά μεγαλύτερη οικειότητα με την διάταξη και την οργάνωση των πληροφοριών σε αυτό.

Έτσι, ο μηχανικός λογισμικού που θέλει να ασχοληθεί με την ΑΜΛ στα Windows είναι σίγουρο πως θα αναγκαστεί πολλές φορές να αναζητήσει μέσα στο ΦΕ τους τομείς που περιέχουν πληροφορίες που αφορούν την αρχιτεκτονική και το λειτουργικό σύστημα για το οποίο σχεδιάστηκε το ΦΕ και ακόμα περισσότερο πληροφορίες για τις δυναμικές βιβλιοθήκες που θα συνδεθούν με το αρχείο. Επίσης πολλές φορές θα κληθεί να επαναφέρει τον πίνακα διευθύνσεων εισαγωγών (Import Address Table) και γι’ αυτό είναι απαραίτητο γι’ αυτόν να κατανοεί πως καταχωρούνται οι πληροφορίες σε αυτόν. Έχοντας αποκτήσει δηλαδή τις βάσεις (για περισσότερα θα πρέπει να ανατρέχει κανείς στην τεκμηρίωση της Microsoft για το πρότυπο PE-COFF) μπορεί να είναι σε θέση κάποιος να κάνει μια τυπική ανάλυση ενός οποιουδήποτε ΦΕ.

Στη συνέχεια θα γίνει μια προσπάθεια να παρουσιαστούν οι βασικές τεχνικές που χρησιμοποιούνται στην ΑΜΛ ώστε στην πορεία να αναλυθεί η λειτουργία μικρών εκτελέσιμων αρχείων χωρίς την ύπαρξη του πηγαίου κώδικα. Στην προσπάθεια να μη δημιουργηθεί κανένα νομικό πρόβλημα στα πλαίσια της εργασίας θα χρησιμοποιηθεί κώδικας από τις αποκαλούμενες “crackme” εφαρμογές που έχουν δημιουργηθεί με μοναδικό στόχο το «σπάσιμο» ή την αντιστροφή της λειτουργίας τους από τους ενδιαφερόμενους να ασχοληθούν με την ΑΜΛ.

4. Βασικές τεχνικές της αντίστροφης μηχανικής λογισμικού και δημοφιλή εργαλεία.

Εισαγωγή

Ξεκινώντας την αντιστροφή ενός εκτελέσιμου αρχείου πρέπει όπως έχει αναφερθεί στο προηγούμενο κεφάλαιο αρχικά να γίνει μια συλλογή όσο το δυνατόν περισσότερων πληροφοριών γίνεται από το ίδιο το αρχείο ελέγχοντας τη δομή του. Στην συνέχεια ο μηχανικός λογισμικού θα χρησιμοποιήσει τεχνικές όπως η **ανακατασκευή αντικειμενικού κώδικα (disassembly)**, **η απόμεταγλώττιση (decompilation)** καθώς και η **αποσφαλμάτωση (debugging)** ανάλογα πάντα με το ΦΕ το οποίο αναλύει καθώς και τις πληροφορίες που χρειάζεται να εξάγει ώστε να επιτύχει τον στόχο του.

Σε αυτό το κεφάλαιο θα αναλυθούν οι βασικές μέθοδοι – τεχνικές που χρησιμοποιούνται, ενώ θα γίνει και αναφορά στα εργαλεία που είχαν τη μεγαλύτερη επίδραση στον χώρο της ΑΜΛ. Ο στόχος είναι η εξοικείωση με τα τυπικά εργαλεία και τις εικόνες που θα βλέπει ο μηχανικός λογισμικού στην προσπάθεια του να αντιστρέψει το εκτελέσιμο.

4.1 Ανακατασκευή αντικειμενικού κώδικα (Disassembly).

Όπως αναφέρθηκε στο 2ο κεφάλαιο, αν κάποιος εξετάσει ένα εκτελέσιμο αρχείο ανοίγοντας το με έναν επεξεργαστή κειμένου που υποστηρίζει αρχεία 16δικού συστήματος (hex editor), θα δει πως το αρχείο αυτό αποτελείται από μια συλλογή από κώδικες λειτουργίας πράγμα το οποίο το καθιστά μη αναγνώσιμο για τον προγραμματιστή. Επομένως το πρώτο στάδιο μετά την ανάλυση της δομής του αρχείου είναι η μετατροπή των κωδικών λειτουργίας σε αναγνώσιμα μνημονικά της συμβολικής γλώσσας. Η διαδικασία αυτή ονομάζεται ανακατασκευή αντικειμενικού ή πηγαίου κώδικα (Disassembly).

Ένα αρχείο πηγαίου κώδικα υψηλού επιπέδου γραμμένο σε κάποια γλώσσα μεταγλωττίζεται σε ένα αντίστοιχο αρχείο συμβολικού κώδικα. Το αρχείο αυτό στη συνέχεια μετασχηματίζεται σε αντικειμενικό αρχείο από έναν συμβολομεταφραστή και ο κώδικας του πλέον είναι άμεσα εκτελέσιμος. Αντικειμενικό αρχείο ή αρχείο αντικειμενικού κώδικα δηλαδή ονομάζουμε ένα αρχείο που περιέχει κώδικα μηχανής αλλά δεν έχει υποστεί σύνδεση, δηλαδή ο συνδέτης (linker) δεν έχει συνενώσει όλα τα αντικειμενικά αρχεία και τις απαραίτητες δυναμικές βιβλιοθήκες σε ένα μοναδικό εκτελέσιμο αρχείο.

Είναι σημαντικό να αναφερθεί πως κατά την ανακατασκευή του κώδικα είναι αδύνατον (για τις περισσότερες γλώσσες προγραμματισμού τουλάχιστον) να επαναφέρει κάποιος τυχόν σχόλια που είχαν προστεθεί στον πηγαίο κώδικα κατά τη δημιουργία της εφαρμογής. Αυτό συμβαίνει γιατί ο μεταγλωττιστής που πιθανόν χρησιμοποιήθηκε ή ο συμβολομεταφραστής τα αφαιρούν στα ενδιάμεσα στάδια μέχρι τη δημιουργία του κώδικα μηχανής.

Πρόκειται για μια διαδικασία αρκετά επίπονη και χρονοβόρα καθώς είναι πολύ εύκολο για κάποιον άνθρωπο να κάνει κάποιο λάθος στην ανάγνωση ενός τμήματος το οποίο θα καταστήσει ολόκληρο τον κώδικα που μεταφράστηκε στην συμβολική γλώσσα μη λειτουργικό. Επίσης γίνεται ακόμα πιο δύσκολη από τη στιγμή που καταβάλλεται περαιτέρω προσπάθεια για να γίνει πιο αναγνώσιμος ο κώδικας από ανθρώπους παρά από μηχανές. Ένα ακόμα στοιχείο που επηρεάζει αυτήν τη διαδικασία είναι η βελτιστοποίηση του κώδικα από τους μεταγλωττιστές. Όταν ο πηγαίος κώδικας μιας εφαρμογής μεταγλωττίζεται, ο μεταγλωττιστής προσπαθεί να βελτιώσει τον κώδικα για

παράδειγμα αφαιρώντας επαναλαμβανόμενα κομμάτια ή αντικαθιστώντας λειτουργικές εντολές με μαθηματικές συναρτήσεις που παράγουν το ίδιο αποτέλεσμα. Στην προσπάθεια όμως για μείωση του όγκου του κώδικα και του χρόνου εκτέλεσης και με τον βαθμό στον οποίο επηρεάζεται ο τελικός αντικειμενικός κώδικας να εξαρτάται άμεσα από την ποιότητα του μεταγλωττιστή δυσχεραίνεται η μετατροπή του στη συμβολική γλώσσα. Στο τέλος απαιτείται πολύ καλή γνώση προγραμματισμού σε χαμηλό επίπεδο ώστε να μπορέσει να παραχθεί ο απαιτούμενος κώδικας και σαφέστατα σημαντικό ρόλο παίζει η εμπειρία και το ένστικτο. Έτσι δημιουργήθηκε η ανάγκη για δημιουργία λογισμικού που θα αυτοματοποιούσε εξ' ολοκλήρου αυτήν τη διαδικασία.

Σαν αποτέλεσμα δημιουργήθηκαν τα βασικότερα εργαλεία της ΑΜΛ για τα οποία χρησιμοποιήθηκε ο όρος «**disassemblers**». Η δουλειά αυτών των εργαλείων ήταν να παρουσιάσουν τις πληροφορίες που «κρύβονταν» μέσα στον χαοτικό κώδικα μηχανής με τρόπο ευανάγνωστο από τους ανθρώπους. Δε θα πρέπει όμως κανείς να νομίζει πως τα εργαλεία αυτά μπορούν να κάνουν την ανάλυση στην θέση του μηχανικού λογισμικού αλλά πρέπει να τα βλέπει μόνο ως εργαλεία παρουσίασης της υπάρχουσας πληροφορίας με κάποια σχετική οργάνωση.

Επίσης σε αυτό το σημείο καλό θα ήταν να διευκρινιστεί πως με τον όρο disassembler αποκαλούνται και εργαλεία που σχεδιάστηκαν για γλώσσες που βασίζονται σε κάποια ενδιάμεση γλώσσα, μεταξύ της γλώσσας υψηλού επιπέδου και της γλώσσας μηχανής, όπως για παράδειγμα παρατηρεί κανείς με την CIL (Common Intermediate Language) στον προγραμματισμό με το .NET Framework όπου ο κώδικας μεταγλωττίζεται από την γλώσσα υψηλού επιπέδου (C#, Visual Basic, Visual C++) σε CIL. Η CIL μπορεί να θεωρηθεί αντίστοιχη με τη συμβολική γλώσσα καθώς είναι το κατώτερο επίπεδο γλώσσας προγραμματισμού που είναι αναγνώσιμη από άνθρωπο χωρίς την ανάγκη όμως για δημιουργία κώδικα μηχανής ενώ το framework αναλαμβάνει να μεταφράσει τις εντολές. Αντίστοιχη είναι και η τεχνολογία που χρησιμοποιεί η Java όπου ο πηγαίος κώδικας μεταγλωττίζεται σε Bytecode ενώ το framework με τη σειρά του αναλαμβάνει την εκτέλεση των εντολών.

Στο Τμήμα Κώδικα 4.1: SimpleFor.cpp φαίνεται ένα τμήμα μιας απλής εφαρμογής γραμμένης σε C++ η οποία θέτει την τιμή 0 στην μεταβλητή «a» και στην συνέχεια αυξάνει

```
int a=0;
for (int i=0; i<10; i++)
{
a++;
}
```

Τμήμα Κώδικα 4.1: SimpleFor.cpp

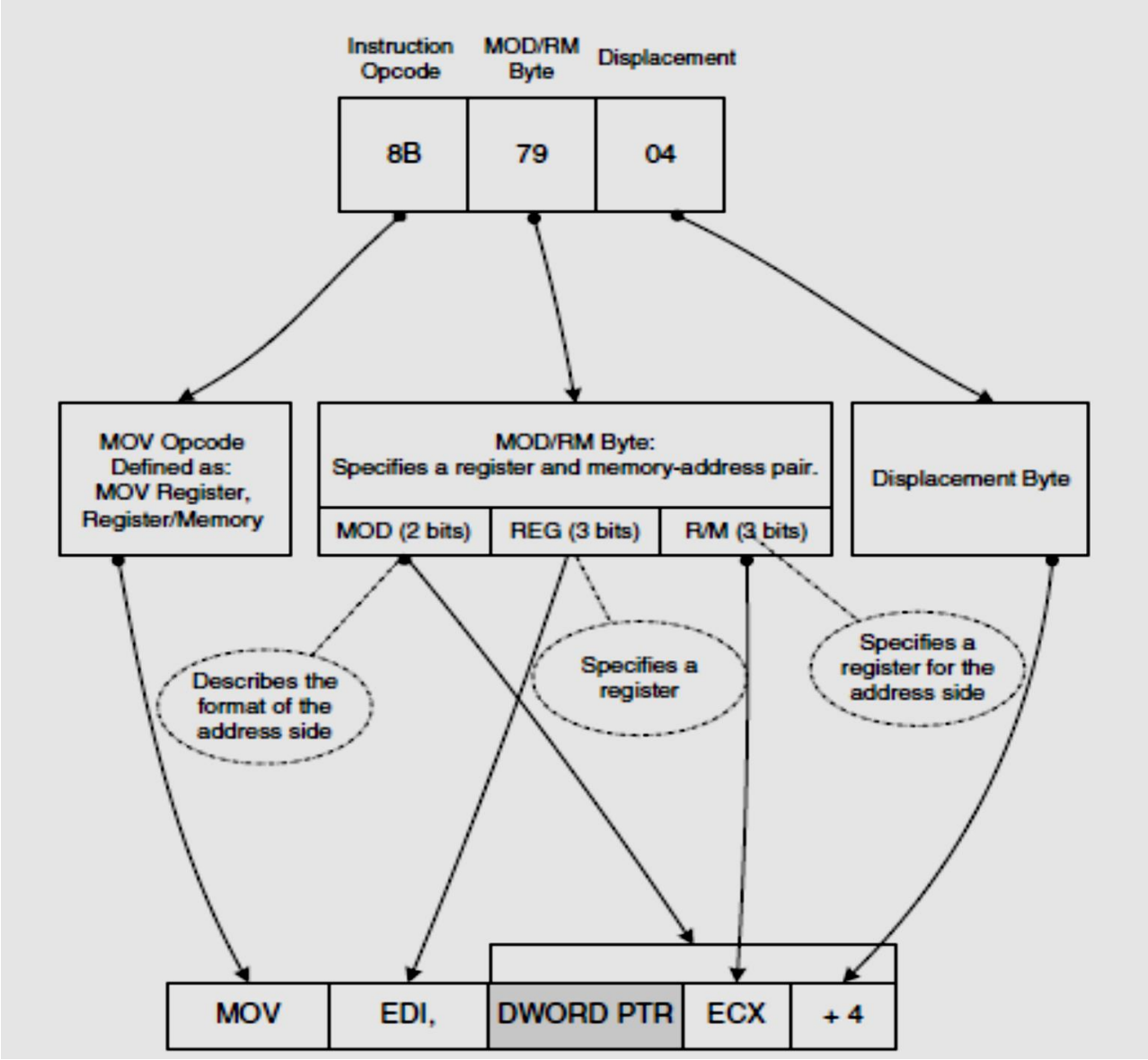
την τιμή της κατά 1, δέκα φορές χρησιμοποιώντας τον μετρητή «i». Το ίδιο τμήμα κώδικα σε έναν disassembler θα φαινόταν όπως στην Εικόνα 4.1: Τμήμα κώδικα σε Disassembler, με την στήλη στα αριστερά να δείχνει την διεύθυνση της εντολής, την κεντρική στήλη την εντολή στο 16δικό σύστημα απευθείας από τον κώδικα μηχανής και την δεξιά στήλη τον κώδικα στη συμβολική γλώσσα. Έτσι διακρίνουμε στην διεύθυνση 0x40131A την ανάθεση της τιμής 0 στην μεταβλητή «a» (mov dword[ebp-0x4], 0x0) και από κάτω την ανάθεση της τιμής 0 στον μετρητή «i» ενώ από την διεύθυνση 0x40132C ως την 0x40133A βλέπουμε την υλοποίηση του επαναληπτικού βρόχου. Τέλος ακολουθούν οι εντολές της συνέχειας της εφαρμογής.

```
L_0040131A:   C7 45 FC 00 00 00 00   mov dword [ebp-0x4], 0x0
L_00401321:   C7 45 F8 00 00 00 00   mov dword [ebp-0x8], 0x0
L_00401328:   83 7D F8 09            cmp dword [ebp-0x8], 0x9
L_0040132C:   7F 0C                  jg 0x40133a
L_0040132E:   8D 45 FC               lea eax, [ebp-0x4]
L_00401331:   FF 00                  inc dword [eax]
L_00401333:   8D 45 F8               lea eax, [ebp-0x8]
L_00401336:   FF 00                  inc dword [eax]
L_00401338:   EB EE                  jmp 0x401328
L_0040133A:   B8 00 00 00 00         mov eax, 0x0
L_0040133F:   C9                     leave
L_00401340:   C3                     ret
```

Εικόνα 4.1: Τμήμα κώδικα σε Disassembler

Για να δημιουργήσουν την έξοδο της προηγούμενης εικόνας, οι disassemblers ακολουθούν μια πολύ απλή διαδικασία για την μετατροπή των εντολών από το εκτελέσιμο αρχείο στη συμβολική γλώσσα. Τυπικά ο disassembler διαβάζει τον κώδικα λειτουργίας (στη συγκεκριμένη περίπτωση 8B) και τον αντιστοιχίζει με κάποιον

πίνακα που περιέχει τους κώδικες λειτουργίας σε αντιστοιχία με τις εντολές της συμβολικής γλώσσας. Στη συνέχεια ανάλογα με την εντολή την οποία χρησιμοποίησε συνεχίζει την ανάγνωση για τους τελεστέους που αυτή περιμένει. Η διαδικασία αυτή παρουσιάζεται καλύτερα στην Εικόνα 4.2: Μετάφραση μιας IA-32 εντολής σε αναγνώσιμο κώδικα συμβολικής γλώσσας (Eilam, 2005).



Εικόνα 4.2: Μετάφραση μιας IA-32 εντολής σε αναγνώσιμο κώδικα συμβολικής γλώσσας (Eilam, 2005)

Ακόμα πολύ ενδιαφέρον παρουσιάζει η μετατροπή του ίδιου κώδικα στην MSIL (CIL όπως την ονομάζει η Microsoft) όπως φαίνεται στην Εικόνα 4.3: Τμήμα κώδικα σε

Disassembler για IL σε IL (Intermediate Language) όπου φαίνεται η διαφορά με την συμβολική γλώσσα. Επειδή όμως η διαδικασία αντιστροφής ενός εκτελέσιμου που βασίζεται στην IL διαφέρει σημαντικά από ότι σε ένα κανονικό εκτελέσιμο αρχείο δε θα γίνει περαιτέρω ανάλυση της IL σε αυτό το σημείο. Ευνόητο όμως είναι πως ανάλογα με τον στόχο προς αντιστροφή ο μηχανικός λογισμικού πρέπει να γνωρίζει όσο το δυνατόν περισσότερα για την γλώσσα χαμηλού επιπέδου που χρησιμοποιείται. Άλλωστε δεν είναι τυχαίο πως ακόμα και ανάμεσα στους πολύ έμπειρους, νόμιμους και μη, που ασχολούνται με την ΑΜΛ υπάρχει διαχωρισμός ανάλογα με την εξειδίκευση τους καθώς είναι αδύνατον να γνωρίζει κάποιος τα πάντα για όλες τις γλώσσες και ειδικά όταν εμπλέκονται οι πολλές τεχνικές για την προστασία του πηγαίου κώδικα.

```
.method assembly static int32 main(string[] args) cil managed
{
  .maxstack 2
  .locals (
    [0] int32 i,
    [1] int32 a,
    [2] int32 num)
  L_0000: ldc.i4.0
  L_0001: stloc.2
  L_0002: ldc.i4.0
  L_0003: stloc.1
  L_0004: ldc.i4.0
  L_0005: stloc.0
  L_0006: br.s L_000c
  L_0008: ldloc.0
  L_0009: ldc.i4.1
  L_000a: add
  L_000b: stloc.0
  L_000c: ldloc.0
  L_000d: ldc.i4.s 10
  L_000f: bge.s L_0017
  L_0011: ldloc.1
  L_0012: ldc.i4.1
  L_0013: add
  L_0014: stloc.1
  L_0015: br.s L_0008
  L_0017: ldc.i4.0
  L_0018: stloc.2
  L_0019: ldloc.2
  L_001a: ret
}
```

Εικόνα 4.3: Τμήμα κώδικα σε Disassembler για IL

4.2 Απομεταγλώττιση (Decompilation)

Η απομεταγλώττιση αποτελεί την δεύτερη μεγάλη κατηγορία στις τεχνικές που χρησιμοποιούνται στην ΑΜΛ. Αντίθετα με την ανακατασκευή του αντικειμενικού κώδικα όπου ο στόχος είναι η δημιουργία κώδικα μιας ενδιάμεσης γλώσσας απλώς αναγνώσιμης από τον άνθρωπο, κατά την απομεταγλώττιση ο στόχος είναι η δημιουργία του αρχικού πηγαίου κώδικα ή όσο το δυνατόν πλησιέστερου κώδικα στον αρχικό που θα έχει την ίδια λειτουργικά. Φυσικά κάποιος κατανοεί πως αυτό θα ήταν το ιδανικό για οποιονδήποτε ήθελε να μάθει περισσότερα για την εφαρμογή που αναλύει όμως ταυτόχρονα είναι και ουτοπικό. Με τις πολλές διαφορετικές πλατφόρμες, αρχιτεκτονικές και τους πάρα πολλούς διαφορετικούς μεταγλωττιστές για μια πληθώρα γλωσσών προγραμματισμού, είναι πρακτικά αδύνατον να μπορεί να ανακατασκευαστεί ο πηγαίος κώδικας για όλα.

Η αλήθεια είναι πως κατά την μεταγλώττιση απαλείφονται πάρα πολλές πληροφορίες με αποτέλεσμα να καθίσταται αδύνατη η επαναφορά του πηγαίου κώδικα για τις περισσότερες γλώσσες. Σε περιπτώσεις όμως όπως η Java και τον προγραμματισμό με το .NET Framework όπου δεν υπάρχει απευθείας μεταγλώττιση σε γλώσσα μηχανής αλλά μόνο σε bytecode και ταυτόχρονα με τη βοήθεια πολλών μετα-δεδομένων που αποθηκεύουν ακόμα περισσότερες χρήσιμες πληροφορίες πλέον γίνεται εφικτό ένα τέτοιο πολύπλοκο εγχείρημα.

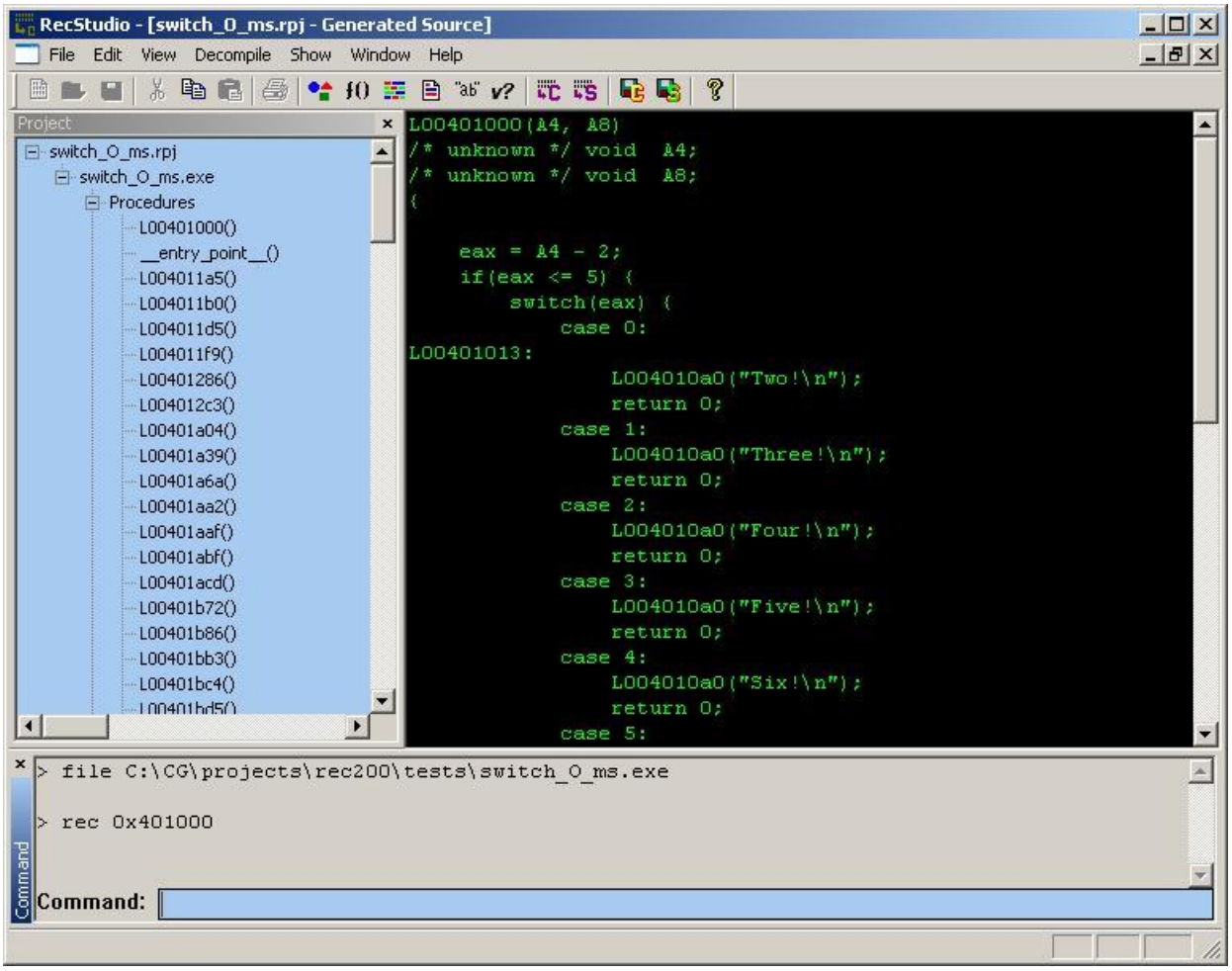
Σίγουρα η δυνατότητα της επαναφοράς του πηγαίου κώδικα είναι θετική από πολλές απόψεις καθώς δεν είναι λίγες οι φορές όπου καταστρέφεται κάποιο τμήμα κώδικα ενώ υπάρχουν λειτουργικές εκδόσεις του εκτελέσιμου αρχείου. Σε αυτές τις περιπτώσεις όταν δεν υπάρχει οργάνωση και αντίγραφα ασφαλείας σίγουρα μπορεί να φανεί σωτήριο όμως στην πραγματικότητα έχει λειτουργήσει ως δίκοπο μαχαίρι καθώς αυτή η δυνατότητα έδωσε στους πειρατές λογισμικού την ευκαιρία να κάνουν τη δουλειά τους πολύ ευκολότερα. Άλλωστε είναι πολύ πιο εύκολο να αλλάξει κάτι σε μια γλώσσα υψηλού επιπέδου ακόμα κι αν δεν υπάρχει η κατάλληλη τεκμηρίωση, παρά να εντοπιστεί το κατάλληλο σημείο για την αλλαγή στη συμβολική γλώσσα. Χαρακτηριστική είναι ακόμα η έκφραση που ακούγεται στην «υπόγεια σκηνή (underground scene)» πως οποιαδήποτε εφαρμογή είναι φτιαγμένη σε .NET μπορεί να «σπάσει».

Όμως είναι πραγματικά έτσι η κατάσταση; Πρέπει να γίνει κατανοητό πως η απομεταγλώττιση δεν είναι πανάκεια και τα εργαλεία δεν μπορούν να ανακαλύψουν

πληροφορία που δεν υπάρχει, επομένως σίγουρα και αυτή η τεχνική παρουσιάζει κάποιους περιορισμούς. Το βασικό πρόβλημα στην απομεταγλώττιση κώδικα μηχανής είναι πως δεν υπάρχει ουσιαστικός τρόπος να αναγνωριστούν οι τύποι δεδομένων και ειδικότερα σύνθετες δομές δεδομένων (structs). Αυτό συμβαίνει γιατί κατά την μεταγλώττιση δεν υπάρχει τρόπος να διατηρηθούν τόσο σύνθετες πληροφορίες. Για παράδειγμα μπορεί να σκεφτεί κάποιος τον αντικειμενοστραφή προγραμματισμό όπου με την κληρονομικότητα και την σύνθεση στις κλάσεις δημιουργούνται πολυσύνθετα αντικείμενα. Το αποτέλεσμα όμως στην συμβολική γλώσσα σίγουρα βασίζεται σε πολύ πιο απλούς τύπους δεδομένων. Βέβαια η λειτουργικότητα παραμένει αλλιώς το πρόγραμμα δεν θα ήταν αυτό που δημιουργήθηκε στην γλώσσα υψηλού επιπέδου όμως σίγουρα είναι λιγότερο κατανοητό. Επομένως ακόμα κι αν κάποιος μπορούσε να επιτύχει την απομεταγλώττιση με σωστό τρόπο, το πρόγραμμα που δημιουργήθηκε σίγουρα θα διατηρούσε την αρχική του λειτουργικότητα αλλά δεν υπάρχει καμιά εγγύηση πως θα ήταν ο ίδιος πηγαίος κώδικας με τον αρχικό ή ότι θα ήταν το ίδιο εύκολα αναγνώσιμος και κατανοητός και φυσικά δεν αναφερόμαστε μόνο στην βασική έλλειψη σχολίων που σίγουρα θα υπήρχαν στον αρχικό κώδικα αλλά και σε ονόματα μεταβλητών ή συναρτήσεων τα οποία σίγουρα θα διευκόλυναν την προσπάθεια κατανόησης του κώδικα.

Παρόλο που το επίπεδο δυνατοτήτων των σημερινών προγραμμάτων απομεταγλώττισης κώδικα μηχανής δεν επαρκεί για την πλήρη απομεταγλώττιση ολόκληρων εφαρμογών, αναμφισβήτητα μπορούν να χρησιμοποιηθούν ως βοηθήματα στην ΑΜΛ. Χαρακτηριστικό παράδειγμα αποτελεί ο Reverse Engineering Compiler (REC - Εικόνα 4.4: Ο απομεταγλωττιστής REC (Backer Street Software, 2007)), που έχει την δυνατότητα μετατροπής κώδικα μηχανής σε μια υποτυπώδη γλώσσα τύπου C με τμήματα κώδικα σε συμβολική γλώσσα. Το αποτέλεσμα της επεξεργασίας με τον REC δεν μπορεί να θεωρηθεί ως πλήρης απομεταγλώττιση της εφαρμογής αλλά σίγουρα είναι πολύ πιο αναγνώσιμο από τον κώδικα σε συμβολική γλώσσα.

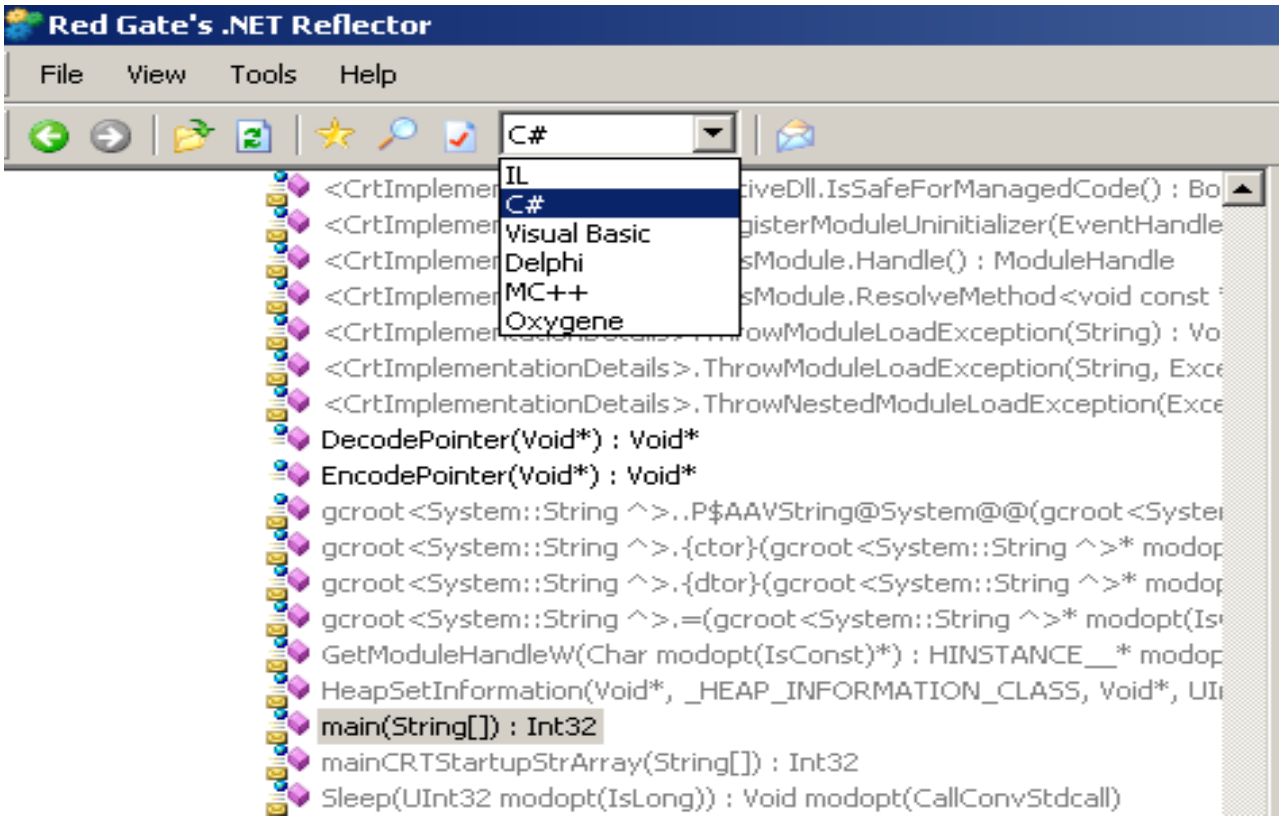
Στον αντίποδα υπάρχουν παραδείγματα πολύ επιτυχημένων εφαρμογών που απομεταγλωττίζουν ενδιάμεσες γλώσσες (IL) και bytecode όπως το πρόγραμμα .NET Reflector της Red Gate τα αποτελέσματα του οποίου είναι αποστομωτικά καθώς μπορεί και απομεταγλωττίζει εφαρμογές γραμμένες σε .NET με χαρακτηριστική ακρίβεια και ευκολία, ενώ παράλληλα δίνει την δυνατότητα απομεταγλώττισης σε όλες τις γλώσσες που υποστηρίζει το .NET Framework και ακόμα και σε Delphi.



Εικόνα 4.4: Ο απομεταγλωττιστής REC (Backer Street Software, 2007)

Όπως θα παρουσιαστεί και στα επόμενα κεφάλαια οι δυνατότητες του αυτές έχουν καθιερώσει το .NET Reflector ως το βασικότερο εργαλείο της ΑΜΛ για την αντιστροφή .NET εφαρμογών. Στην Εικόνα 4.5: Net Reflector ονόματα συναρτήσεων - μεθόδων παρουσιάζονται τα ονόματα των συναρτήσεων που βρέθηκαν μετά την επεξεργασία στο **Error! Reference source not found.** γραμμένο σε Visual C++ και μετά την ανάλυση του πό το .NET Reflector. Ο χρήστης μπορεί πολύ εύκολα να ανακαλύψει την κεντρική μέθοδο – main() και διαλέγοντας την γλώσσα που επιθυμεί μπορεί να δει στην συνέχεια τον πηγαίο κώδικα που συνθέτει αυτή τη μέθοδο – συνάρτηση όπως φαίνεται καλύτερα στην . Αμέσως μπορεί να παρατηρήσει κάποιος ότι ο κώδικας έχει υποστεί κάποια αλλοίωση με την προσθήκη λέξεων όπως **«internal static»** πριν την μέθοδο και την αντικατάσταση της

έκφρασης **«i++»** με το ισοδύναμο **«i = (int) (I + 1)»**. Το βασικό στοιχείο που



Εικόνα 4.5: Net Reflector ονόματα συναρτήσεων - μεθόδων

προκύπτει είναι πως λόγω των μεταδεδομένων που αποθηκεύονται στα ΦΕ του .NET ο απομεταγλωττιστής είναι σε θέση να διακρίνει τα ονόματα των συναρτήσεων και των μεταβλητών όμως ο πηγαίος κώδικας που παράγεται δεν είναι ταυτόσημος. Το αποτέλεσμα όπως απεικονίζεται στην Εικόνα 4.6 αποτελεί μια παραλλαγή του αυθεντικού πηγαίου κώδικα που όμως δεν διαφέρει καθόλου σε λειτουργικότητα από τον αρχικό.

Disassembler

```
internal static int main(string[] args)
{
    int a = 0;
    for (int i = 0; i < 10; i = (int) (i + 1))
    {
        a = (int) (a + 1);
    }
    return 0;
}
```

Εικόνα 4.6: .NET Reflector πηγαίος κώδικας επιλεγμένης μεθόδου

4.3 Αποσφαλμάτωση (Debugging)

Το βασικό χαρακτηριστικό των δύο προηγούμενων τεχνικών είναι η στατική όψη του ΦΕ που παρέχουν στον μηχανικό λογισμικού. Με τον όρο «στατική» περιγράφεται η κατάσταση στην οποία η παρατήρηση του ΦΕ και η ανάλυση του γίνεται με βάση τη μορφή που αυτό έχει όπως είναι αποθηκευμένο στο δίσκο όμως πολλές φορές αυτό δεν είναι αρκετό. Αρκεί μόνο να φανταστεί κάποιος πως μια εφαρμογή μπορεί να αποτελείται από εκατομμύρια γραμμές κώδικα με αποτέλεσμα να είναι πολύ επίπονη διαδικασία να ανακαλύψει κανείς το σημείο στο οποίο πραγματικά αναλύεται η διαδικασία την οποία παρατηρεί όταν λειτουργεί την εφαρμογή. Επομένως μετά από αρκετά χρόνια εμφανίστηκε η ανάγκη της παρατήρησης του κώδικα δυναμικά, δηλαδή καθώς αυτός εκτελείται από το λειτουργικό σύστημα.

Πραγματικά στα πρώτα χρόνια της ΑΜΛ όλα γίνονταν με τη χρήση ενός disassembler, χαρτιού και μολυβιού αλλά καθώς εμφανίζονταν ολοκληρωμένα περιβάλλοντα ανάπτυξης (Integrated Development Environment – IDE) που παρείχαν τη δυνατότητα της αποσφαλμάτωσης, ήταν αναπόφευκτο να γίνουν τεράστιες προσπάθειες για την δημιουργία λογισμικού που θα επέτρεπε την αποσφαλμάτωση κώδικα μηχανής. Ο όρος αποσφαλμάτωση υποδηλώνει την δυνατότητα που παρέχεται στον προγραμματιστή να εκτελεί τον κώδικα της εφαρμογής του γραμμή με γραμμή, βήμα προς βήμα και με την πρόσθετη δυνατότητα να σταματάει την ροή της εκτέλεσης εκεί που αυτός επιθυμεί ενώ ταυτόχρονα να μπορεί να δει την κατάσταση της μνήμης και των καταχωρητών τη στιγμή εκείνη ώστε να μπορέσει να κατανοήσει καλύτερα που βρίσκονται τα τυχόν σφάλματα στον κώδικα του και να τα διορθώσει.

Η διαφορά όμως με την ΑΜΛ και με ένα IDE είναι πως στην μεν δεν υπάρχει ο πηγαίος κώδικας ενώ στην άλλη περίπτωση υπάρχει. Αυτή η διαφορά αποτέλεσε τροχοπέδη στην προσπάθεια για δημιουργία ενός εργαλείου αποσφαλμάτωσης (debugger) για εκτελέσιμα αρχεία σε κώδικα μηχανής. Ακόμα, έπρεπε να λυθεί και το πρόβλημα της απεικόνισης του κώδικα μηχανής σε κατανοητή μορφή και να παρέχεται η δυνατότητα παρατήρησης της μνήμης σε κάθε σημείο της εκτέλεσης. Επίσης έπρεπε ο μηχανικός λογισμικού να μπορεί να τροποποιεί το ΦΕ κατά βούληση. Σαν αποτέλεσμα, άρχισαν να δημιουργούνται εργαλεία που αποτελούσαν έναν συνδυασμό ενός ισχυρού disassembler με εργαλεία παρακολούθησης της κατάστασης της μνήμης και των

καταχωρητών. Ακόμα, αργότερα προστέθηκε και η δυνατότητα επέμβασης στις τιμές των καταχωρητών, της μνήμης και επεξεργασίας του ΦΕ δίνοντας στον μηχανικό λογισμικού την δυνατότητα απόλυτου ελέγχου της ροής εκτέλεσης της εφαρμογής.

Συνοψίζοντας θα μπορούσε κανείς να πει πως ένας debugger είναι το εργαλείο που βοηθά και καθοδηγεί τον μηχανικό λογισμικού ώστε να αντιστρέψει μόνο τα πραγματικά ενδιαφέροντα σημεία του κώδικα και όχι ολόκληρη την εφαρμογή. Εξάλλου είναι πιο πιθανό σενάριο χρήσης να χρειάζεται να τροποποιηθεί ένα τμήμα του κώδικα από ότι ολόκληρη η εφαρμογή.

4.3.1 Βασικά χαρακτηριστικά ενός Debugger.

Ένας debugger όπως προαναφέραμε αποτελείται από μια πληθώρα εργαλείων παρακολούθησης σε συνδυασμό με δυνατότητες στατικής ανάλυσης του κώδικα. Παρόλο όμως που υπάρχουν σημαντικές διαφορές στις δυνατότητες που παρέχουν, ένας debugger βασίζεται στις παρακάτω βασικές δυνατότητες.

Σημεία διακοπής (Breakpoints)

Το κυριότερο χαρακτηριστικό ενός debugger είναι η δυνατότητα που παρέχει στον χρήστη να εισάγει σημεία διακοπής στον κώδικα που εκτελείται, δηλαδή εισάγει την ώρα της εκτέλεσης πρόσθετο κώδικα ο οποίος όταν εκτελεστεί δίνει τον έλεγχο στον χρήστη και παγώνει την εκτέλεση του προγράμματος. Υπάρχουν δύο είδη σημείων διακοπής, τα σημεία διακοπής υλικού και λογισμικού, hardware breakpoints και software breakpoints αντίστοιχα.

Τα σημεία διακοπής υλικού όπως δηλώνει και το όνομα τους απαιτούν υποστήριξη από το υλικό του υπολογιστή και συγκεκριμένα από την κεντρική μονάδα επεξεργασίας (ΚΜΕ). Η ΚΜΕ στην αρχιτεκτονική IA-32 προσφέρει τη δυνατότητα διακοπών υλικού με την ταυτόχρονη χρήση 4 καταχωρητών αποσφαλμάτωσης (Debug Registers). Οι καταχωρητές αυτοί αριθμούν από D0 εώς D3 και έχουν μέγεθος 1 byte. Για να εφαρμοστεί ένα σημείο διακοπής απαιτούνται 1, 2 ή 4 bytes ανάλογα με το είδος της διακοπής. Τα 4 είδη διακοπών υλικού που επιτρέπονται είναι τα παρακάτω:

1. **Διακοπή κατά την εκτέλεση (Break on execution)** όπου ο έλεγχος επαναφέρεται στον χρήστη (άμεσα στον debugger και έμμεσα στον χρήστη) όταν εκτελεστεί κάποιο τμήμα κώδικα. Οι διακοπές κατά την εκτέλεση απαιτούν 1 byte.
2. **Διακοπή κατά την πρόσβαση στην μνήμη (Break on memory access)** όπου ο έλεγχος επαναφέρεται στον χρήστη όταν υπάρξει πρόσβαση σε κάποια θέση μνήμης συμπεριλαμβανομένης της ανάγνωσης από και της εγγραφής σε αυτήν. Αυτού του τύπου οι διακοπές είναι πολύ χρήσιμες στην αναγνώριση τμημάτων κώδικα η δεδομένων στο ΦΕ. Ο χρήστης μπορεί να εντοπίσει τα τμήματα κώδικα που επηρεάζουν τα δεδομένα πολύ γρήγορα και εύκολα εισάγοντας ένα σημείο διακοπής κατά την πρόσβαση στη μνήμη για την διεύθυνση που περιέχει τα δεδομένα που τον ενδιαφέρουν.
3. **Διακοπή κατά την εγγραφή στην μνήμη (Break on memory write)** όπου ο έλεγχος επαναφέρεται στον χρήστη κατά την τροποποίηση της τιμής μιας θέσης μνήμης.
4. **Διακοπή κατά την πρόσβαση σε θύρες I/O (Break on I/O port access).** Όπως δηλώνει το όνομα επαναφέρει τον έλεγχο στον χρήστη όταν υπάρξει κάποια ενέργεια σε κάποια θύρα εισόδου/εξόδου. Πολλές φορές δεν υπάρχει στους debuggers γιατί χρησιμοποιείται πολύ σπάνια. Συνήθως χρησιμοποιείται κατά την αντιστροφή οδηγών συσκευών.

Σε αυτό το σημείο καλό είναι να επισημανθεί πως στην αρχιτεκτονική IA-32 επιτρέπεται η χρήση μόνο 4 σημείων διακοπής υλικού ενώ σε ΚΜΕ x64 επεκτάθηκαν σε 8.

Αντίθετα, τα σημεία διακοπής λογισμικού αποτελούν μια εξομοίωση της λειτουργίας των σημείων διακοπής υλικού κατά την εκτέλεση. Η εξομοίωση είναι μια πολύ απλή διαδικασία καθώς ο debugger αντικαθιστά την εντολή προς εκτέλεση με μια εντολή που παγιδεύει την εκτέλεση (συνήθως INT 3). Ταυτόχρονα αποθηκεύει την αυθεντική εντολή και την επαναφέρει ως την επόμενη προς εκτέλεση εντολή όταν συνεχιστεί η εκτέλεση του κώδικα. Επιτρέπεται στον χρήστη να εισάγει όσα σημεία διακοπής λογισμικού θέλει ενώ παράλληλα βλέπει τον κώδικα με την προηγούμενη μορφή του, δηλαδή τα σημεία αυτά είναι αόρατα στον χρήστη.

Disassembler

Πολύ σημαντικό χαρακτηριστικό ενός καλού εργαλείου αποσφαλμάτωσης είναι η ύπαρξη ενός πολύ καλού disassembler. Πρέπει να προσφέρει στον χρήστη τη δυνατότητα τόσο αυτόματης στατικής ανάλυσης του κώδικα όσο και τη δυνατότητα να παρουσιάζει τον κώδικα ως έχει (πολύ χρήσιμο για πολυμορφικά αρχεία). Ακόμα είναι πολύ σημαντικό να μπορεί να κρατάει τις σημειώσεις του χρήστη στη μνήμη.

Οπτική αναπαράσταση των καταχωρητών και της μνήμης

Ακόμα πολύ σημαντικό είναι για τον χρήστη να βλέπει την τρέχουσα κατάσταση των καταχωρητών, της μνήμης και ειδικότερα του σωρού. Επίσης θα πρέπει να του παρέχεται η δυνατότητα να μπορεί να τροποποιεί τις τιμές σε κάθε σημείο της μνήμης καθώς και των καταχωρητών ώστε να μπορεί να αλλάζει την ροή εκτέλεσης κατά βούληση και να βλέπει εν δράσει την επιρροή τυχόν αλλαγών σε τιμές της μνήμης. Τέλος, αν και σπάνια παρέχεται σαν δυνατότητα, πολύ ενδιαφέρον παρουσιάζει μια προεπισκόπηση της κατάστασης των καταχωρητών κατά την εκτέλεση της επόμενης στη σειρά εντολής.

Λεπτομέρειες που αφορούν την διεργασία, τα νήματα της και τις μονάδες (modules)

Τελευταίο αλλά εξίσου σημαντικό για τον χρήστη είναι να έχει πληροφορίες για τυχόν νήματα που δημιουργούνται από την διεργασία που εξετάζει, να μπορεί να επεξεργαστεί αυτά τα νήματα και ακόμα να του δίνονται πληροφορίες που αφορούν τις μονάδες (π.χ. αρχεία DLL) οι οποίες χρησιμοποιούνται ανά πάσα στιγμή από την διεργασία. Οποιοδήποτε αρχείο φορτώνεται στην μνήμη σαν αποτέλεσμα της εκτέλεσης του ΦΕ θα πρέπει να εξετάζεται και να δίνεται στον χρήστη η δυνατότητα της εμφάνισης του άμεσα στον disassembler του debugger.

4.3.2 Είδη εργαλείων αποσφαλμάτωσης

Οι debuggers χωρίζονται σε δύο κατηγορίες ανάλογα με το επίπεδο στο οποίο δρουν. Οι user-mode debuggers όπως υποδηλώνει και το όνομα δουλεύουν στο επίπεδο του χρήστη, δηλαδή δουλεύουν με τον ίδιο τρόπο όπως κάθε άλλη εφαρμογή. Εμφανίζονται ως διεργασίες και έχουν πρόσβαση σε περιοχές μνήμης που είναι διαθέσιμες για τις διεργασίες του χρήστη. Έτσι για παράδειγμα δεν μπορούν να «διαβάσουν» την περιοχή της μνήμης που δεσμεύεται από το λειτουργικό σύστημα ή από οδηγούς συσκευών. Από την άλλη πλευρά οι kernel-mode debuggers όπως δηλώνει το όνομα τους λειτουργούν στο επίπεδο του πυρήνα, δηλαδή πιο χαμηλά από τον χρήστη. Σαν αποτέλεσμα έχουν πολύ περισσότερες δυνατότητες, μπορούν να χρησιμοποιηθούν για την αποσφαλμάτωση οδηγών συσκευών ή του ίδιου του λειτουργικού συστήματος και ουσιαστικά δίνουν μεγαλύτερη αυτονομία στον χρήστη.

Μπορεί όμως κάποιος εύκολα να αναρωτηθεί σε τι παραπάνω μπορεί να βοηθήσει κάποιον μηχανικό λογισμικού αυτή η δυνατότητα και γιατί να επιλέξει κάποιον kernel-mode debugger αν δεν στοχεύει στην αποσφαλμάτωση του λειτουργικού συστήματος. Η απάντηση γίνεται εμφανής αν κάποιος διανοηθεί πως η βασική μέθοδος ανάλυσης κακόβουλου λογισμικού είναι η ΑΜΛ. Αν κάποιος συνδυάσει αυτήν την ιδέα με το γεγονός πως κάθε ΦΕ ή μονάδα μπορεί να κρύβει κακόβουλο κώδικα τότε αν αυτό δρα σε χαμηλότερο επίπεδο από τον χρήστη το αποτέλεσμα είναι να μην επιτρέπεται σε έναν user-mode debugger η επεξεργασία του. Έτσι αμέσως δημιουργείται ένας μεγάλος περιορισμός στην ασφάλεια ενός συστήματος ή τουλάχιστον στην προσπάθεια που καταβάλλεται καθημερινά για δημιουργία περισσότερο ασφαλών συστημάτων.

«Τα Rootkit (κακόβουλο λογισμικό που επιτρέπει την συνεχή πρόσβαση σε έναν υπολογιστή με προνόμια υπερχρήστη, ενώ κρύβει ενεργά την παρουσία του από τους διαχειριστές με το να ενσωματώνεται σε βασικά αρχεία του λειτουργικού συστήματος ή άλλων εφαρμογών) έχουν αναπτυχθεί εδώ και χρόνια ως οδηγοί συσκευών. Κακόβουλο λογισμικό πλέον δημιουργείται χρησιμοποιώντας την ίδια τεχνική και ενσωματώνοντας τον εαυτό του στον πυρήνα. Το λογισμικό DRM (Digital Rights Management) που χρησιμοποιείται για την προστασία της πνευματικής ιδιοκτησίας σε ψηφιακά δεδομένα πολλές φορές περιέχει οδηγούς συσκευών με ευάλωτο κώδικα όπως αποδείχτηκε από το CVE-2007-5887 (Kaminsky, Ferguson, Larsen, Miras, & Pearce, 2008)».

4.3.3 Δημοφιλή εργαλεία αποσφαλμάτωσης

Στα επόμενα κεφάλαια οι αναλύσεις των ΦΕ θα γίνονται με user-mode debuggers και κυρίως με τους δύο πιο δημοφιλείς, Olly Debugger και IDA Pro και για αυτό θα γίνει μια σύντομη παρουσίαση τους όπως και για τον διάσημο kernel-mode debugger της Numega τον SoftIce που για πάρα πολλά χρόνια κυριάρχησε στην ΑΜΛ.

Numega SoftIce

Δεν θα ήταν καθόλου υπερβολή να θεωρήσει κανείς τον SoftIce ως το πιο διαδεδομένο εργαλείο αποσφαλμάτωσης καθώς για πολλά χρόνια ήταν το προτιμώμενο εργαλείο για πάρα πολλούς έμπειρους μηχανικούς λογισμικού που ασχολούνταν με την ΑΜΛ. Το βασικό χαρακτηριστικό του είναι πως πρόκειται για kernel-mode debugger δηλαδή επιτρέπει την αποσφαλμάτωση του τοπικού πυρήνα του λειτουργικού συστήματος. Αρχικά ξεκίνησε ως εργαλείο για την ανάπτυξη οδηγών συσκευών αλλά πολύ γρήγορα άρχισε να χρησιμοποιείται ως εργαλείο της ΑΜΛ εξαιτίας αυτής του της ιδιότητας.

Σε αντίθεση με τους user-mode debuggers που θα δούμε στην συνέχεια, ο SoftIce δίνει τη δυνατότητα παύσης της λειτουργίας της εφαρμογής-στόχου σε οποιοδήποτε σημείο καθώς με το πάτημα ενός συνδυασμού πλήκτρων σταματάει τον πυρήνα και εμφανίζει πληροφορίες σχετικά με την κατάσταση όλου του λειτουργικού συστήματος, δηλαδή έμμεσα και της εφαρμογής-στόχου. Σημειωτέον πως εκείνη τη στιγμή οι πληροφορίες παρουσιάζονται σε ένα παράθυρο το οποίο δεν ανήκει στα windows καθώς αυτά έχουν «παγώσει» από τον SoftIce ο οποίος αναλαμβάνει εξ’ ολοκλήρου την διαχείριση του. Επίσης δεν επιτρέπει τις διακοπές (interrupts) για κανέναν λόγο και σε συστήματα με πολλούς πυρήνες οι πολλούς επεξεργαστές αναλαμβάνει πλήρως τον έλεγχο όλων των επεξεργαστών.

Βασικό μειονέκτημα του SoftIce είναι η αποσταθεροποίηση του συστήματος. Η επιρροή στο σύστημα γίνεται παρατηρήσιμη πολλές φορές καθώς αυτή η ενσωμάτωση του στον πυρήνα αναγκάζει πολλές φορές το λειτουργικό σύστημα να αστοχήσει ή να βρεθεί σε αδιέξοδο (κατάσταση όπου οι διεργασίες δεν μπορούν να ζητήσουν τους

πόρους που χρειάζονται για να συνεχίσουν καθώς αυτοί είναι δεσμευμένοι από άλλες διεργασίες) με αποτέλεσμα την γνωστή σε όλους μπλε οθόνη θανάτου (Blue Screen Of Death – BSOD). Ακόμα πολλές εφαρμογές αρνούνται να τρέξουν σε περίπτωση που υπάρχει kernel-mode debugger εγκατεστημένος. Τέλος, υπάρχουν πολλές ασυμβατότητες με άλλους οδηγούς συσκευών που επηρεάζουν τον πυρήνα του λειτουργικού συστήματος.

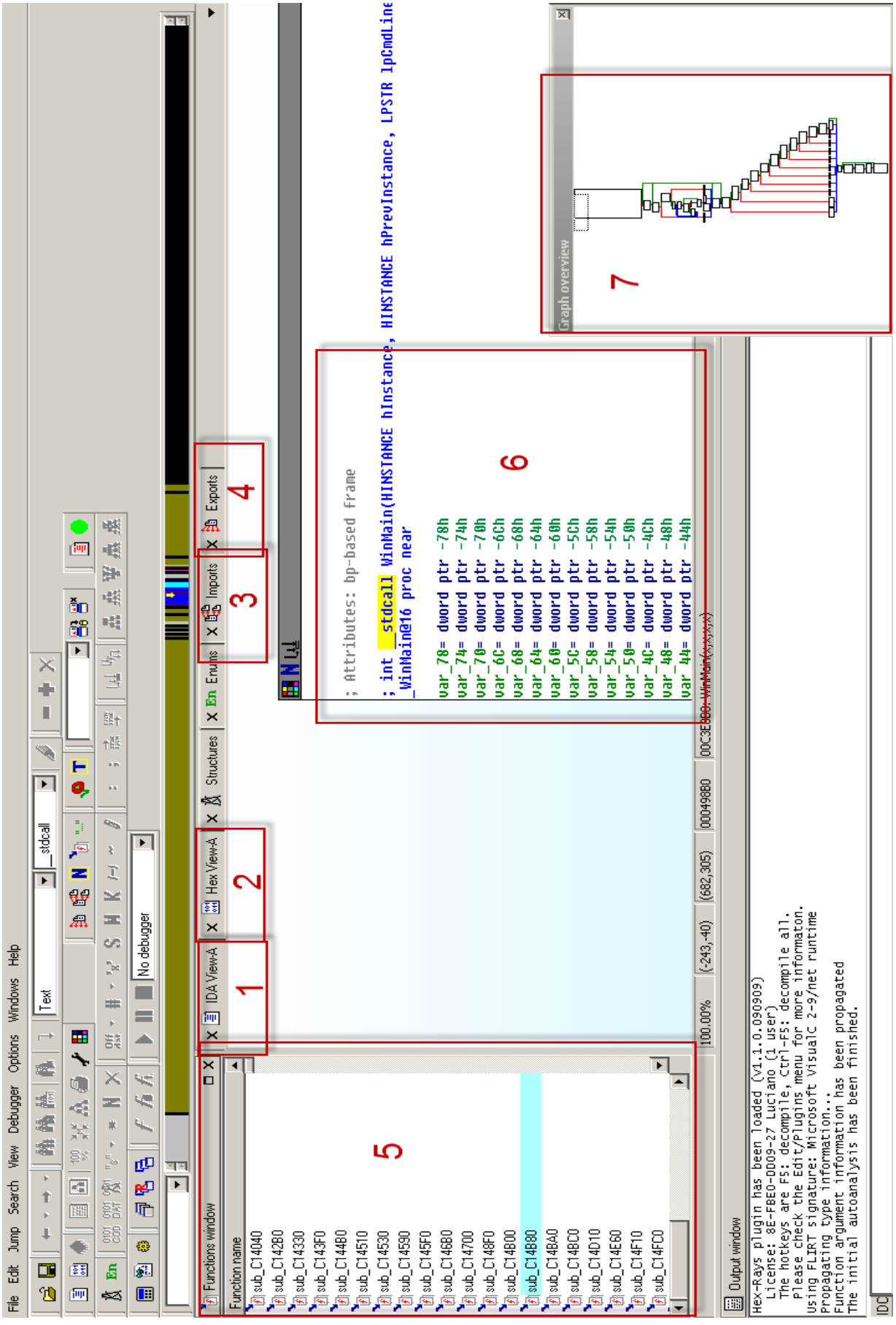
Το κύριο πλεονέκτημα του SoftIce είναι πως μπορεί κανείς να δει πως επηρεάζεται ολόκληρο το λειτουργικό σύστημα από μια εφαρμογή ενώ επιτρέπει και να βλέπει κανείς τις διεργασίες που εκτελούνται σαν σύνολο και όχι σαν απλές οντότητες. Εξάλλου πολλές φορές οι εφαρμογές βασίζονται σε περισσότερες από δυο διεργασίες οι οποίες επικοινωνούν μεταξύ τους με κάποιον αόρατο για τον χρήστη τρόπο.

Συμπερασματικά μπορεί κανείς να καταλάβει γιατί ο SoftIce γρήγορα έγινε το αγαπημένο εργαλείο των μηχανικών λογισμικού εξαιτίας των πολύπλευρων δυνατοτήτων του. Πριν από μερικά χρόνια όμως η Numega σταμάτησε την υποστήριξη του SoftIce με αποτέλεσμα να παραμείνει στάσιμος και σιγά σιγά αχρηστεύθηκε. Ακόμα με την ύπαρξη των Windows Vista και των Windows 7 η θέση του έγινε ακόμα πιο δύσκολη λόγω ασυμβατότητας. Παρόλα αυτά ακόμα και σήμερα μπορεί να χρησιμοποιηθεί σε περιπτώσεις που απαιτείται αποσφαλμάτωση στο επίπεδο του πυρήνα χρησιμοποιώντας εικονικά περιβάλλοντα όπως το VMWare και το Virtual PC. Σίγουρα όμως παρόλο που έχει περιοριστεί αρκετά η προσφορά του στον χώρο της ΑΜΛ του προσφέρει μια θέση άξια αναφοράς έστω και για ιστορικούς λόγους.

IDA Pro

Καθώς στα πλαίσια της εργασίας θα χρησιμοποιηθούν user-mode debuggers μπορούμε να πούμε πως ο IDA Pro είναι ένα από τα πιο χρήσιμα εργαλεία που δημιουργήθηκαν ποτέ για την ΑΜΛ. Στην ουσία αντί για εργαλείο αποσφαλμάτωσης θα μπορούσαμε να το αποκαλέσουμε καλύτερα μια σουίτα αποσφαλμάτωσης λόγω της πληθώρας των δυνατοτήτων του μερικές από τις οποίες θα εξερευνηθούν μελετώντας αρχικά την διεπαφή (Interface) του IDA Pro όπως αυτή απεικονίζεται στην Εικόνα 4.7: Διεπαφή (Interface) του IDA Pro. Τα ιδιαίτερα σημεία ενδιαφέροντος έχουν σημειωθεί με αριθμούς. Αναλυτικότερα:

1. **IDA View-A:** Κάνοντας κλικ σε αυτήν την καρτέλα εμφανίζεται το τμήμα 6 που είναι η αναπαράσταση του disassembler του IDA Pro.
2. **Hex View-A:** Κάνοντας κλικ σε αυτήν την καρτέλα θα εμφανιστεί στο τμήμα 6 η αναπαράσταση του κώδικα μηχανής που βρίσκεται στο ΦΕ στο 16δικό σύστημα καθώς και σε ASCII μορφή.
3. **Imports:** Κάνοντας κλικ σε αυτήν την καρτέλα ο χρήστης μπορεί να δει μια λίστα με όλες τις συναρτήσεις από δυναμικές βιβλιοθήκες ή μονάδες που εισάγονται κατά την φόρτωση του αρχείου με χρήση των πληροφοριών που βρίσκονται στον πίνακα διευθύνσεων εισαγωγών (IAT).
4. **Exports:** Κάνοντας κλικ σε αυτήν την καρτέλα ο χρήστης μπορεί να δει μια λίστα με όλες τις συναρτήσεις που μπορούν να εξαχθούν από το αρχείο με χρήση των πληροφοριών που βρίσκονται στον πίνακα διευθύνσεων εξαγωγών (EAT).
5. **Functions Windows:** Σε αυτό το παράθυρο ο χρήστης βλέπει τα ονόματα όλων των συναρτήσεων ή καλύτερα των υπορουτινών που αναγνωρίστηκαν κατά την ανάλυση του ΦΕ από τον disassembler του IDA Pro. Σε περίπτωση που δεν πρόκειται για γλώσσα bytecode τότε οι υπορουτίνες ονομάζονται με τη χρήση του προθέματος sub (subroutine) και της 16δικής διεύθυνσης από την οποία ξεκινούν.
6. **Hex View-A Window:** Σε αυτό το παράθυρο εμφανίζεται ένα είδος γραφήματος της ροής εκτέλεσης του ΦΕ για τις υπορουτίνες που συνεργάζονται μεταξύ τους. Στην συγκεκριμένη περίπτωση παρουσιάζεται η υπορουτίνα που βρίσκεται στο σημείο εισόδου του ΦΕ. Κάποιος μπορεί να περιηγηθεί στο παράθυρο, να δει τις πληροφορίες που περιέχει η υπορουτίνα και τις δυνατές εναλλακτικές ροές εκτέλεσης. Αυτή η παρουσίαση της πληροφορίας μπορεί να βοηθήσει τον μηχανικό λογισμικού να κατανοήσει καλύτερα πως λειτουργεί η εφαρμογή σαν να ήταν ένα διάγραμμα ροής δεδομένων. Αυτό ακριβώς το χαρακτηριστικό του IDA Pro τον έχει κάνει πολύ δημοφιλή στους κύκλους της ΑΜΛ καθώς πολλές φορές μπορεί ο μηχανικός λογισμικού να ανακαλύψει πολύ γρήγορα οπτικά την σωστή ροή εκτέλεσης.
7. **Graph Overview:** Πρόκειται για μια προεπισκόπηση του παραθύρου IDA View-A την οποία μπορεί ο χρήστης να χρησιμοποιήσει για να δει καλύτερα το γράφημα πιο ολοκληρωμένο και να περιηγηθεί σε αυτό.



Εικόνα 4.7: Διεπαφή (Interface) του IDA Pro

Όμως οι δυνατότητες του IDA Pro δεν περιορίζονται στην πολύ καλή διεπαφή του και τον πολύ δυνατό disassembler του. Ένα άλλο πολύ σημαντικό στοιχείο είναι η δυνατότητα του να επεξεργάζεται πολλούς τύπους αρχείων για πολλές πλατφόρμες και λειτουργικά συστήματα όπως για Unix, Mac, Handheld PC, PDA και κονσόλες παιχνιδιών. Επίσης χρήσιμη θα αποδειχθεί η δυνατότητα του να αναζητά συμβολοσειρές. Επομένως στην πραγματικότητα πρόκειται για ένα πολυεργαλείο με ελάχιστους περιορισμούς. Ακόμα επιτρέπει στον χρήστη να τρέχει δικά του scripts γεγονός που επιτρέπει εξειδικευμένες επεξεργασίες του ΦΕ όπως είναι η αφαίρεση απόκρυψης κώδικα σε .NET αρχεία.

Τα βασικά μειονεκτήματα του είναι η έλλειψη δυνατότητας επεξεργασίας του ΦΕ και η αδυναμία του απόλυτου ελέγχου του κατά την αποσφαλμάτωση καθώς και το γεγονός ότι δεν παρέχεται δωρεάν η ολοκληρωμένη έκδοση του.

Olly Debugger (OllyDbg)

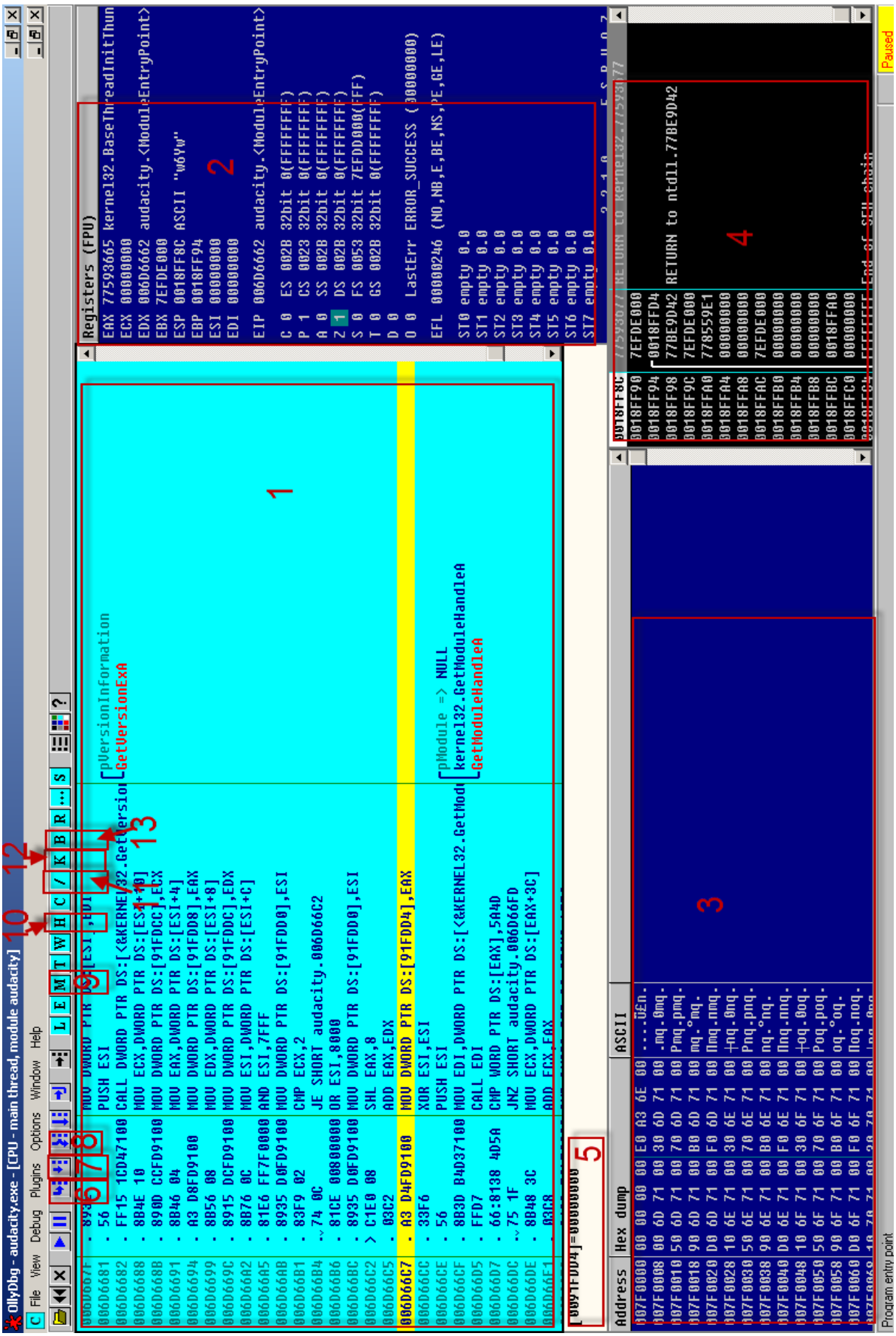
Ένα ακόμη από τα βασικά εργαλεία που θα χρησιμοποιηθούν στα πλαίσια της εργασίας είναι ο Olly Debugger (OllyDbg) γνωστός και ως απλά Olly. Ο Olly σχεδιάστηκε από την αρχή σαν εργαλείο της ΑΜΛ γι’ αυτό έχει δοθεί μεγάλη προσοχή στις δυνατότητες του για ανακατασκευή κώδικα. Έτσι ο disassembler του παρέχει πάρα πολλές δυνατότητες ανάλυσης και αναγνώρισης πολύπλοκων δομών, εναλλακτικών ροών και βρόχων επανάληψης. Ακόμα μπορεί και αναλύει τον κώδικα πολύ εύκολα βρίσκοντας ακόμα και τα ονόματα για τις εισαγόμενες συναρτήσεις από δυναμικές βιβλιοθήκες όταν υπάρχουν σύμβολα. Στη συνέχεια ο χρήστης αν έχει στην κατοχή του την κατάλληλη τεκμηρίωση σαν Help 2.0 αρχείο της Microsoft μπορεί να την ενοποιήσει με τον Olly και να αναζητά περισσότερες πληροφορίες για τις συναρτήσεις που ανακαλύφθηκαν. Χαρακτηριστικό παράδειγμα αποτελεί το αρχείο βοήθειας για το Win32 API το οποίο αποτελεί την πιο βασική πηγή πληροφόρησης για τον μηχανικό λογισμικού. Μια άλλη πολύ χρήσιμη δυνατότητα του Olly είναι ο ενσωματωμένος επεξεργαστής συμβολικής γλώσσας που δίνει στον μηχανικό λογισμικού την δυνατότητα να γράφει κώδικα, να τον εισάγει μέσα στο εκτελέσιμο αρχείο στη θέση που θέλει και είτε να τον αποθηκεύει μόνιμα στο εκτελέσιμο είτε προσωρινά στην μνήμη του Olly. Ο Olly στην συνέχεια μπορεί να αντικαθιστά τα τμήματα που περιέχουν τις διορθώσεις (patches) κατά την εκτέλεση χωρίς να επηρεάζει το ίδιο το εκτελέσιμο παρά μόνο για εκείνη την εκτέλεση μέχρι ο μηχανικός λογισμικού να

αποφασίσει να αποθηκεύσει μόνιμα τις αλλαγές του. Ακόμα μπορεί κανείς με τον Olly να δει ανά πάσα στιγμή και για οποιοδήποτε τμήμα κώδικα τις αναφορές από και προς το τμήμα αυτό. Τέλος πέρα από τις δυνατότητες τροποποίησης καταχωρητών, μνήμης, σωρού και κώδικα πολύ σημαντική θα αποδειχθεί στη συνέχεια η δυνατότητα του για χρήση επεκτάσεων (plugins), ειδικότερα όταν θα χρειαστεί να αποκρύψουμε τον Olly από την εφαρμογή-στόχο.

Ουσιαστικά δηλαδή ο Olly χαρακτηρίζεται κυρίως από το γεγονός πως είναι ένα πλήρως παραμετροποιήσιμο εργαλείο αποσφαλμάτωσης και κυκλοφορεί σε πολλές εκδόσεις προ-ρυθμισμένες από άλλους μηχανικούς λογισμικού ώστε να «ξεγελούν» τυποποιημένες εμπορικές εφαρμογές προστασίας λογισμικού και από την ευχρηστία του. Συνήθως άτομα που ξεκινούν να μαθαίνουν τις τεχνικές της ΑΜΛ προσπαθούν να χρησιμοποιήσουν πολύπλοκα εργαλεία όπως ο SoftIce και μετά την αποτυχία τους (λόγω της πολυπλοκότητας του εργαλείου αυτού) σταματούν την προσπάθεια τους, ενώ αντίθετα με τον Olly οι πληροφορίες τους παρέχονται με πιο εύκολο και κατανοητό τρόπο. Στην Εικόνα 4.8: Διεπαφή (Interface) του OllyDbg παρουσιάζεται ο τρόπος με τον οποίο είναι οργανωμένη η διεπαφή του Olly. Τα σημεία ενδιαφέροντος έχουν σημειωθεί με αριθμούς. Αναλυτικότερα:

1. Στο παράθυρο αυτό εμφανίζεται ο κώδικας του εκτελέσιμου όπως τον παρουσιάζει ο disassembler. Στην αριστερή στήλη βρίσκεται η διεύθυνση στη μνήμη, στη μεσαία ο κώδικας λειτουργίας και στη δεξιά η ίδια έκφραση στην συμβολική γλώσσα. Ακόμα ο Olly εμφανίζει κάποια σημάδια αριστερά από τις εντολές και υποδεικνύει τις αλλαγές στις ροές ή τις διάφορες υπορουτίνες. Ο χρήστης έχει δυνατότητα να επιλέξει οποιαδήποτε γραμμή κώδικα και να την τροποποιήσει όπως αυτός επιθυμεί.
2. Στο παράθυρο αυτό εμφανίζονται όλοι οι καταχωρητές γενικής χρήσης και οι βασικοί καταχωρητές σημαιών. Ο χρήστης έχει τη δυνατότητα να εμφανίζει και άλλους καταχωρητές όπως της FPU, τους καταχωρητές MMX και τους καταχωρητές αποσφαλμάτωσης (έγινε αναφορά στα σημεία διακοπής υλικού). Ο χρήστης έχει τη δυνατότητα να αλλάζει την τιμή τους και να επηρεάζει έμμεσα την ροή της εκτέλεσης του κώδικα.
3. **Dump Window:** Σε αυτό το παράθυρο απεικονίζεται ο κώδικας του εκτελέσιμου αρχείου στο 16δικό σύστημα καθώς και σε ASCII μορφή.

4. **Stack Window:** Σε αυτό το παράθυρο εμφανίζεται ο σωρός στη μνήμη με τις διευθύνσεις και τις αντίστοιχες τιμές.
5. Σε αυτό το παράθυρο εμφανίζεται η έκφραση του επιλεγμένου κώδικα στον disassembler με το αποτέλεσμα της.
6. **Step Into:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εκτελεί την αμέσως επόμενη εντολή και σε περίπτωση που πρόκειται για κλήση σε κάποια υπορουτίνα εισέρχεται σε αυτήν και εκτελεί την πρώτη εντολή της.
7. **Step Over:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εκτελεί την αμέσως επόμενη εντολή. Σε περίπτωση που πρόκειται για κλήση σε κάποια υπορουτίνα ολοκληρώνει την εκτέλεση της.
8. **Trace Into:** Κάνοντας κλικ στο κουμπί αυτό ο Olly ιχνηλατεί όλο τον κώδικα και τον αναλύει. Παράλληλα ιχνηλατεί και κάθε κλήση σε υπορουτίνες.
9. **Memory Map:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εμφανίζει έναν χάρτη της μνήμης.
10. **Handles:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εμφανίζει τα handles προς οποιαδήποτε οντότητα χρησιμοποιείται από την εφαρμογή.
11. **Patches:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εμφανίζει τις διορθώσεις κώδικα (patches) που έχει αποθηκεύσει προσωρινά ο χρήστης. Κατά την εκτέλεση του κώδικα σε πραγματικό χρόνο ο Olly θα χρησιμοποιήσει τις πληροφορίες αυτές για να προβεί στις αντικαταστάσεις.
12. **Call Stack:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εμφανίζει μια λίστα με τη μορφή σωρού από όλες τις κλήσεις σε υπορουτίνες που έχουν γίνει μέχρι το σημείο που έχει εκτελεστεί ο κώδικας τη δεδομένη στιγμή.
13. **Breakpoints:** Κάνοντας κλικ στο κουμπί αυτό ο Olly εμφανίζει τη λίστα με όλα τα σημεία διακοπής που έχει εισάγει ο χρήστης στον κώδικα. Μπορεί στη συνέχεια να απενεργοποιήσει κάποια από αυτά αν θελήσει ή να τα διαγράψει εντελώς από τη λίστα.



Εικόνα 4.8: Διεπαφή (Interface) του OllyDbg

Έτσι από την διεπαφή και μόνο μπορεί να καταλάβει κάποιος το πλήθος των πληροφοριών που γίνονται διαθέσιμες πολύ εύκολα στον χρήστη. Οι δυνατότητες του Olly όμως δεν περιορίζονται σε αυτά τα χαρακτηριστικά μόνο καθώς με τη δυνατότητα εισαγωγής επεκτάσεων μπορεί να τροποποιηθεί σημαντικά η διεπαφή ανάλογα με τις ανάγκες του χρήστη και τις απαιτήσεις της εφαρμογής-στόχου.

Ο μοναδικός περιορισμός του Olly μπορεί κανείς να πει πως είναι η έλλειψη δυνατότητας επεξεργασίας εφαρμογών 64bit και το γεγονός πως έχει παρουσιάσει πολλά ευάλωτα σημεία στον κώδικα του με αποτέλεσμα οι προγραμματιστές να τα εκμεταλλεύονται και να αναγκάζουν τον Olly σε αστοχίες κατά την προσπάθεια αποσφαλμάτωσης ορισμένων εφαρμογών. Το γεγονός αυτό σε συνδυασμό με την λήξη της υποστήριξης από τον αρχικό συγγραφέα του ο οποίος ασχολείται αποκλειστικά με την ανάπτυξη της δεύτερης έκδοσης του πετυχημένου εργαλείου όμως δεν απέτρεψε την κοινότητα του από το να δημιουργεί επεκτάσεις που διορθώνουν τα ευάλωτα σημεία αυτά.

4.4 Σύνοψη

Στην ενότητα αυτή αναλύθηκαν οι τρεις βασικές μέθοδοι ανάλυσης ενός εκτελέσιμου αρχείου της ΑΜΛ. Πιο συγκεκριμένα έγινε αναφορά στην προσπάθεια ανακατασκευής του κώδικα μηχανής σε συμβολική γλώσσα (disassembly), στον τρόπο με τον οποίο αυτή επιτυγχάνεται και τους περιορισμούς που υπάρχουν. Ακόμα έγινε αναφορά στην δυσκολία της δημιουργίας ενός εργαλείου που θα μπορούσε να πετύχει ανακατασκευή του κώδικα μηχανής σε γλώσσα υψηλού επιπέδου, δηλαδή την απομεταγλώττιση ενός εκτελέσιμου αρχείου. Παρουσιάστηκαν οι λόγοι για τους οποίους αυτό είναι εφικτό μόνο σε γλώσσες που χρησιμοποιούν κάποιο είδος bytecode αλλά ακόμα και σε αυτές υπήρξε δυσκολία πλήρους επαναφοράς του κώδικα. Το κύριο χαρακτηριστικό και των δύο τεχνικών αυτών ήταν πως βασίζονταν στην στατική ανάλυση του κώδικα. Αντίθετα, στο τέλος έγινε εκτενής αναφορά στην επικρατούσα τεχνική της ΑΜΛ, την αποσφαλμάτωση, η οποία επιτρέπει την δυναμική ανάλυση μιας εφαρμογής.

Πέρα όμως από την ανάλυση των τεχνικών έγινε μια προσπάθεια να παρουσιαστούν τα βασικά εργαλεία αποσφαλμάτωσης που θα χρησιμοποιηθούν στην εργασία, ο IDA Pro και ο Olly Debugger. Στόχος ήταν η εξοικείωση του μηχανικού λογισμικού με την διεπαφή (interface) των εργαλείων αυτών και ο κατατοπισμός σχετικά με την οργάνωση των πληροφοριών σε αυτήν πριν την εισαγωγή σε δυσκολότερες έννοιες όπως είναι η ανίχνευση του κώδικα, οι αλλαγές στη ροή εκτέλεσης και οι διορθώσεις στον κώδικα μηχανής οι οποίες θα παρουσιαστούν αναλυτικά στο επόμενο κεφάλαιο.

5. Η αντίστροφη μηχανική λογισμικού και η παράκαμψη τυπικών μέτρων προστασίας εφαρμογών.

Εισαγωγή

Σε πολλές εμπορικές εφαρμογές λογισμικού δίνεται η δυνατότητα στον χρήστη να τις χρησιμοποιήσει για κάποιο διάστημα ή με κάποιους περιορισμούς ελεύθερα για επιδεικτικούς σκοπούς. Για να μπορέσει να επιτευχθεί ο διαφημιστικός αυτός στόχος όμως πρέπει να εισαχθούν κάποια μέτρα προστασίας. Αυτά τα μέτρα μπορεί να περιλαμβάνουν απόκρυψη του κώδικα, σειριακούς αριθμούς, περιορισμούς λειτουργιών, ενοχλητικά μηνύματα, διαφημίσεις και προτροπές στον χρήστη να αγοράσει το προϊόν. Σε αυτό όμως το σημείο παρουσιάζει ενδιαφέρον πώς κάποιοι διαρρήκτες λογισμικού (crackers) καταφέρνουν χρησιμοποιώντας τεχνικές της ΑΜΛ να τροποποιούν το λογισμικό ώστε να λειτουργεί κανονικά σαν να μην ήταν προστατευμένο.

Έτσι λοιπόν στο κεφάλαιο αυτό θα γίνει μια προσπάθεια να παρουσιαστούν οι διάφορες μεθοδολογίες που ακολουθούνται τόσο από τις εταιρείες που παράγουν το λογισμικό όσο και από τους διαρρήκτες λογισμικού. Παράλληλα, κατά τη μελέτη αυτών των τεχνικών, οι γνώσεις που αποκτήθηκαν από τα προηγούμενα κεφάλαια θα αποτελέσουν το θεωρητικό υπόβαθρο ενώ με το τέλος του κεφαλαίου θα έχει γίνει πιο διακριτή η συσχέτιση του θεωρητικού μέρους της εργασίας με την πρακτική εφαρμογή στον πραγματικό κόσμο.

Στην προσπάθεια να μη δημιουργηθεί πρόβλημα, από εδώ και πέρα στην εργασία θα χρησιμοποιηθούν οι αποκαλούμενες “crackme” εφαρμογές που έχουν σχεδιαστεί ακριβώς για να μπορούν οι μηχανικοί λογισμικού να δοκιμάζουν τις ικανότητες τους στην ΑΜΛ και εφαρμογές που έχουν γραφτεί ειδικά για την εργασία αυτή.

5.1 Ανίχνευση (Tracing) και ροή εκτέλεσης

Με τον όρο ανίχνευση στην ΑΜΛ αναφερόμαστε σε μια ειδικότερη διαδικασία ανάλυσης του κώδικα κατά την εκτέλεση του. Η ανάλυση αυτή ανάλογα το επίπεδο στο οποίο συμβαίνει καταγράφει και διαφορετικές πληροφορίες. Το βασικό επίπεδο στο οποίο μπορεί να γίνει ανίχνευση είναι το επίπεδο των εντολών. Σε αυτό το σημείο καταγράφεται μετά την εκτέλεση κάθε εντολής η τρέχουσα κατάσταση των καταχωρητών. Ευνόητο είναι όμως πως μια τέτοια διαδικασία μπορεί να αποβεί απείρως χρονοβόρα ανάλογα με το μέγεθος της εφαρμογής και έτσι παρόλο που παρέχει τη μέγιστη δυνατή πληροφόρηση καθίσταται ανίκανη να βοηθήσει σημαντικά τον μηχανικό λογισμικού. Αντί αυτής μπορεί να εφαρμοστεί μια μέθοδος ανάλυσης σε επίπεδο συναρτήσεων (υπορουτινών) όπου θα καταγράφονται πληροφορίες που αφορούν τα ορίσματα τα οποία εισάγονται σε κάθε συνάρτηση και προαιρετικά οι καταστάσεις των καταχωρητών κατά την κλήση. Μια ακόμα σημαντική πληροφορία είναι η καταγραφή της συνάρτησης από την οποία προήλθε η κλήση. Το πλεονέκτημα από αυτήν την μέθοδο προκύπτει βασικά από την μείωση του χρόνου που απαιτείται για την επεξεργασία των πληροφοριών όμως πολλές φορές και αυτή αποτυγχάνει να δώσει την απαραίτητη πληροφόρηση στον μηχανικό λογισμικού.

Σε μια ιδανική κατάσταση ο μηχανικός λογισμικού επιθυμεί την αναγνώριση βασικών δομικών τμημάτων μέσα στον κώδικα τα οποία δεν έχουν διακλαδώσεις, δηλαδή δεν υπάρχουν εναλλακτικές ροές εκτέλεσης μέσα σε ένα τμήμα κώδικα. Αυτού του είδους η ανίχνευση είναι άκρως σημαντική για δύο βασικούς λόγους:

1. Οι περισσότερες τροποποιήσεις σε ένα εκτελέσιμο αρχείο αφορούν τροποποίηση της ροής εκτέλεσης του προγράμματος και επομένως είναι κρίσιμης σημασίας ο μηχανικός λογισμικού να γνωρίζει το πώς και το γιατί η εφαρμογή ακολουθεί μία συγκεκριμένη ροή εκτέλεσης.
2. Επειδή πολλές φορές δεν είναι εφικτή η άμεση αλλαγή της ροής εκτέλεσης, ο μηχανικός λογισμικού χρειάζεται περισσότερες πληροφορίες ώστε να μπορεί να τροποποιήσει την είσοδο κάποιων συναρτήσεων ώστε να προκαλέσει την επιθυμητή αλλαγή στην ροή εκτέλεσης.

Η σημασία της ανίχνευσης θα φαίνεται σε κάθε βήμα καθώς θα παρουσιάζονται πιο προχωρημένες έννοιες της ΑΜΛ.

5.2 Απλή επιδιόρθωση κώδικα (patching).

Η απλή επιδιόρθωση κώδικα (patching) αποτελεί την πιο βασική μορφή των επεμβάσεων σε μια εφαρμογή. Ο μηχανικός λογισμικού εντοπίζει τον προβληματικό κώδικα αναλύοντας την εφαρμογή-στόχο με τις προαναφερθείσες μεθόδους της ΑΜΛ και τον τροποποιεί εισάγοντας τον δικό του επιδιορθωμένο κώδικα. Συνήθως ο στόχος είναι η επίτευξη μικρών αλλαγών στην ροή εκτέλεσης του κώδικα της εφαρμογής. Χαρακτηριστικά παραδείγματα αποτελούν εφαρμογές οι οποίες εφόσον δεν έχουν αγοραστεί ακόμα εμφανίζουν διάφορα διαφημιστικά ή γενικότερα ενοχλητικά μηνύματα (nag screens) στον χρήστη και εφαρμογές οι οποίες ζητούν όνομα χρήστη και συνθηματικό για να επιτρέψουν την πρόσβαση στο περιβάλλον τους. Τέτοιες εφαρμογές βασίζονται σε κάποιο τμήμα κώδικα λογικού ελέγχου – για παράδειγμα μια εντολή «IF» θα μπορούσε να ελέγχει την ταυτοποίηση ενός κωδικού χρήστη για την είσοδο του στο σύστημα – και ανάλογα επιλέγεται είτε η «σωστή» είτε η «λανθασμένη» ροή εκτέλεσης.

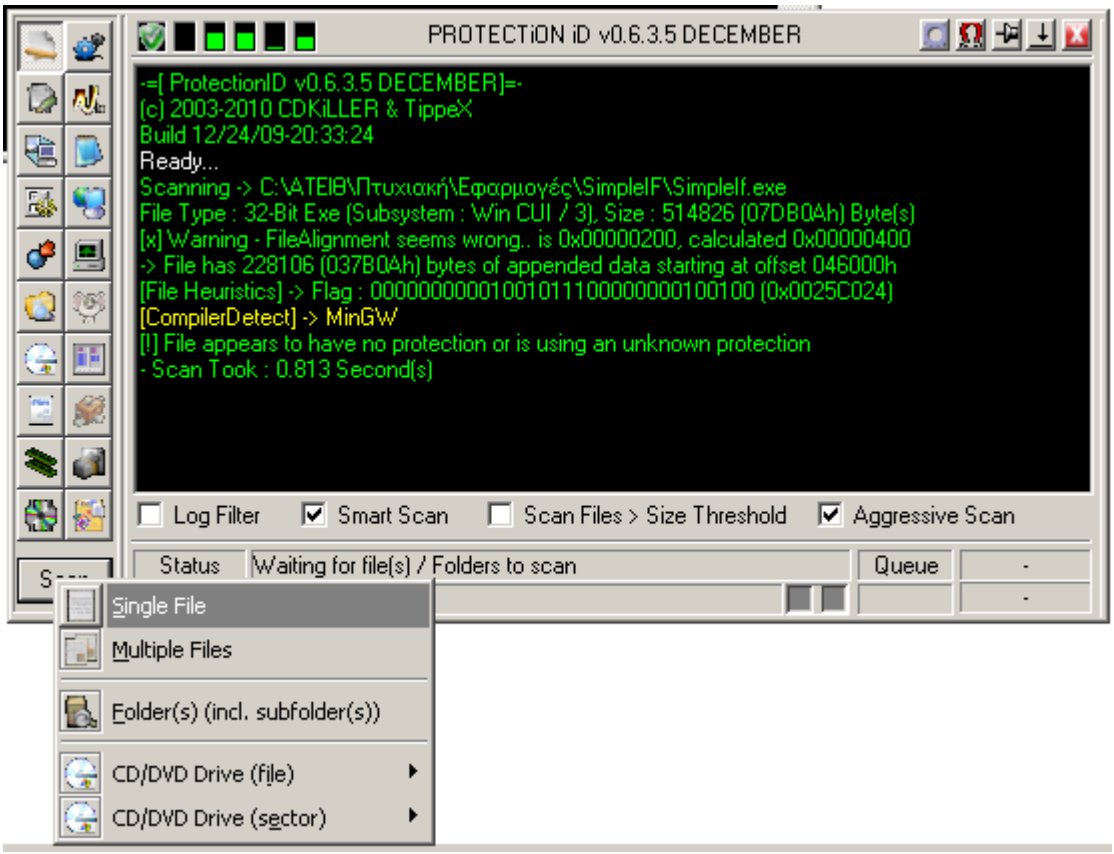
Αυτό θα φανεί στην προσπάθεια να επιτευχθεί αλλαγή στην ροή εκτέλεσης στο εκτελέσιμο αρχείο της Εφαρμογή SimpleIF.cpp (βλ. Παράρτημα Α).

5.2.1 Ανάλυση της εφαρμογής

ΒΗΜΑ 1ο

Όπως και για κάθε εφαρμογή-στόχο θα πρέπει να γίνει μια αρχική ανάλυση της εφαρμογής. Μερικές πληροφορίες μπορούν να ανευρεθούν εύκολα χρησιμοποιώντας το πρόγραμμα Protection ID (CDKiLLER & TippeX, © 2004-2011) που παρουσιάζεται στην Εικόνα 5.1. Με την ολοκλήρωση της αρχικής ανάλυσης του αρχείου όπως φαίνεται στην Εικόνα 5.2 μπορεί κανείς να δει ήδη πως η εφαρμογή είναι φτιαγμένη για πλατφόρμες 32bit καθώς και ότι έχει δημιουργηθεί από τον μεταγλωττιστή MinGW γνωστό μεταγλωττιστή της C++ και επίσης δίνει μια εκτίμηση για την μέθοδο προστασίας της εφαρμογής. Στη συγκεκριμένη εφαρμογή δεν μπόρεσε να ανιχνεύσει κάποια γνωστή μέθοδο προστασίας.

Εικόνα 5.1: Πληροφορίες για το Protection ID



Εικόνα 5.2: Πληροφορίες της εφαρμογής SimpleIF

ΒΗΜΑ 2ο

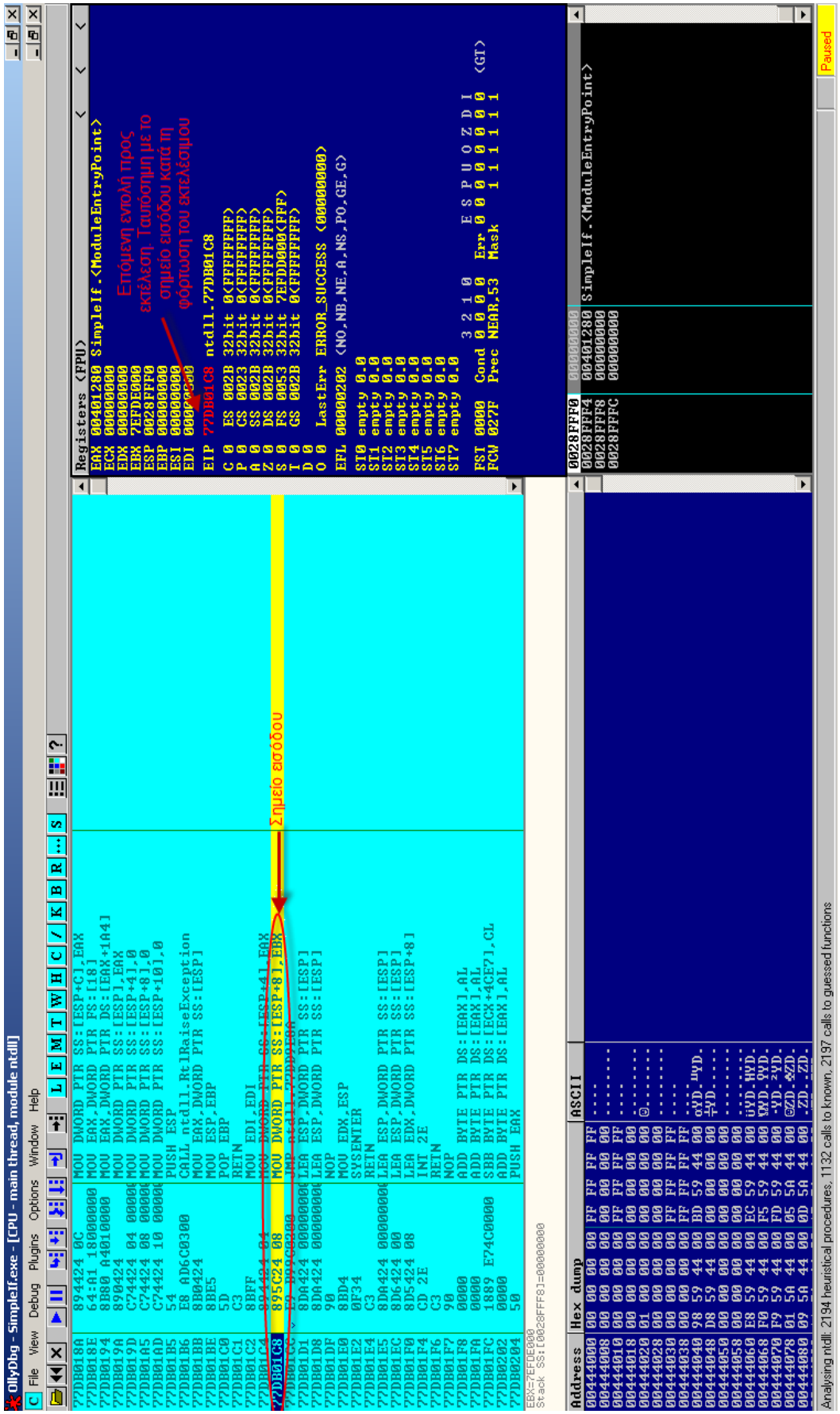
Η εφαρμογή πρέπει να φορτωθεί στον OllyDbg ο οποίος θα παρέχει περισσότερες πληροφορίες και επίσης θα βοηθήσει στην αναγνώριση των κρίσιμων ροών εκτέλεσης της εφαρμογής. Έτσι άμεσα με την φόρτωση της εφαρμογής ο OllyDbg παγώνει την εκτέλεση της εφαρμογής στο σημείο εισόδου του εκτελέσιμου αρχείου. Στη συγκεκριμένη περίπτωση όπως φαίνεται στην Εικόνα 5.3, βλέπουμε πως η εκτέλεση έχει σταματήσει στο σημείο εισόδου του «module ntdll» δηλαδή στην πρώτη εντολή που θα εκτελεστεί από το «ntdll» αλλά ο έλεγχος έχει μεταφερθεί στον OllyDbg. Φυσικά ο χρήστης ελέγχει τον OllyDbg και επομένως και το εκτελέσιμο. Ακόμα είναι σημαντική πληροφορία η τιμή που έχει πάρει ο καταχωρητής EAX και αποτελεί το σημείο εισόδου για τον πραγματικό κώδικα της εφαρμογής.

ΒΗΜΑ 3ο

Ο μηχανικός πρέπει να εκτελέσει την εφαρμογή βήμα προς βήμα και να παρατηρήσει την συμπεριφορά της, δηλαδή να σημειώσει κάθε σημείο στο οποίο έγινε αλλαγή της ροής εκτέλεσης ή διάφορα μηνύματα που εμφανίστηκαν κατά την εκτέλεση. Αυτό είναι μια επίπονη και χρονοβόρα διαδικασία και πολλές φορές μπορεί να χρειαστεί να επανεκτελεστεί η εφαρμογή ώστε να γίνει καλύτερη μελέτη και σε μεγαλύτερο βάθος κάποιον κλήσεων σε υπορουτίνες (αντί για step over χρειάζεται να γίνει step into από τη διεπαφή του OllyDbg) με στόχο να εντοπιστούν οι κρίσιμες δομές της εφαρμογής. Πολλές φορές ίσως χρειαστεί να δημιουργηθούν διαγράμματα για να αποσαφηνιστούν κάποιες από αυτές τις κλήσεις. Έτσι στην συγκεκριμένη εφαρμογή παρατηρεί πως εμφανίζονται κάποια βασικά μηνύματα για την εισαγωγή του αναγνωριστικού χρήστη (username) και του κωδικού πρόσβασης (password) του. Τέλος όταν αυτά εισαχθούν λανθασμένα εμφανίζεται ένα μήνυμα λάθους και μια προειδοποίηση τερματισμού της εφαρμογής.

Το πρώτο ενδιαφέρον σημείο αποτελεί το παρακάτω τμήμα κώδικα και πιο

```
00401283   .  83EC 08          SUB ESP,8
00401286   .  C70424 01000000   MOV DWORD PTR SS:[ESP],1
0040128D   .  FF15 04E24400     CALL DWORD PTR DS:[<&msvcrt.__set_app_type>;
00401293   .  E8 B8FEFFFF       CALL SimpleIf.00401150
```



Εικόνα 5.3: Ο OllyDbg μεταφέρει τον έλεγχο της εφαρμογής στον χρήστη

συγκεκριμένα η τελευταία εντολή παρατηρεί κανείς ότι όταν εκτελεστεί (step over) με την κλήση της ρουτίνας «CALL SimpleIf.00401150» επιτρέπει σχεδόν όλη την κανονική λειτουργία της εφαρμογής μέχρι την εμφάνιση του μηνύματος λάθους. Σαν αποτέλεσμα μπορεί κανείς να συμπεράνει πως αποτελεί τον πυρήνα της εφαρμογής και μέσα της γίνονται όλες οι λειτουργίες ταυτοποίησης του χρήστη. Συνεπώς το πρώτο σημείο διακοπής λογισμικού θα πρέπει να τοποθετηθεί στην διεύθυνση «00401293» στο συγκεκριμένο παράδειγμα (οι διευθύνσεις μπορεί να διαφέρουν σε κάθε μηχάνημα). Επομένως θα πρέπει να γίνει «step into» για να παρατηρηθεί περαιτέρω η συμπεριφορά της εφαρμογής. Με τον ίδιο τρόπο ανακαλύπτουμε το παρακάτω ενδιαφέρον τμήμα κώδικα.

```
0040123E  |.  A1 04804400       MOV EAX,DWORD PTR DS:[448004]
00401243  |.  890424            MOV DWORD PTR SS:[ESP],EAX
00401246  |.  E8 A3010000       CALL SimpleIf.004013EE
0040124B  |.  89C3              MOV EBX,EAX                    ; |
0040124D  |.  E8 66460100       CALL <JMP.&msvcrt._cexit>              ; |[msvcrt._cexit
```

Εδώ παρατηρούμε ακριβώς την ίδια συμπεριφορά κατά την εκτέλεση της εντολής «CALL SimpleIf.004013EE» όπως προηγουμένως και έτσι περιορίζουμε ακόμα περισσότερο το ενδιαφέρον σημείο της εφαρμογής εισάγοντας ένα καινούριο σημείο διακοπής λογισμικού. Καθώς συνεχίζεται η εκτέλεση με «step into» εντοπίζουμε σε διάφορα σημεία εντολές που αφορούν κάποιες συμβολοσειρές τις οποίες μεταφράζει ο OllyDbg σε αναγνώσιμο ASCII κώδικα. Έτσι βλέπουμε εντολές όπως:

```
00401465  .  C74424 04 00504400 MOV DWORD PTR SS:[ESP+4],SimpleIf.00445000      ;
ASCII "admin"
004014DC  .  C74424 04 06504400 MOV DWORD PTR SS:[ESP+4],SimpleIf.00445006      ;
ASCII "adminPassword"
00401558  .  C74424 04 14504400 MOV DWORD PTR SS:[ESP+4],SimpleIf.00445014      ;
ASCII "Please enter your username"
00401560  .  C70424 C0834400    MOV DWORD PTR SS:[ESP],SimpleIf.004483C0        ;  ASCII
"DrD"
```

ενώ η εκτέλεση σταματά στην γραμμή

```
00401594  .  E8 C7070400       CALL SimpleIf.00441D60
```

όπου η εφαρμογή σταματά και περιμένει την είσοδο του χρήστη για να συνεχίσει. Τέλος παρατηρεί κανείς ότι συμβαίνει το ίδιο και για τον κωδικό χρήστη στις εντολές

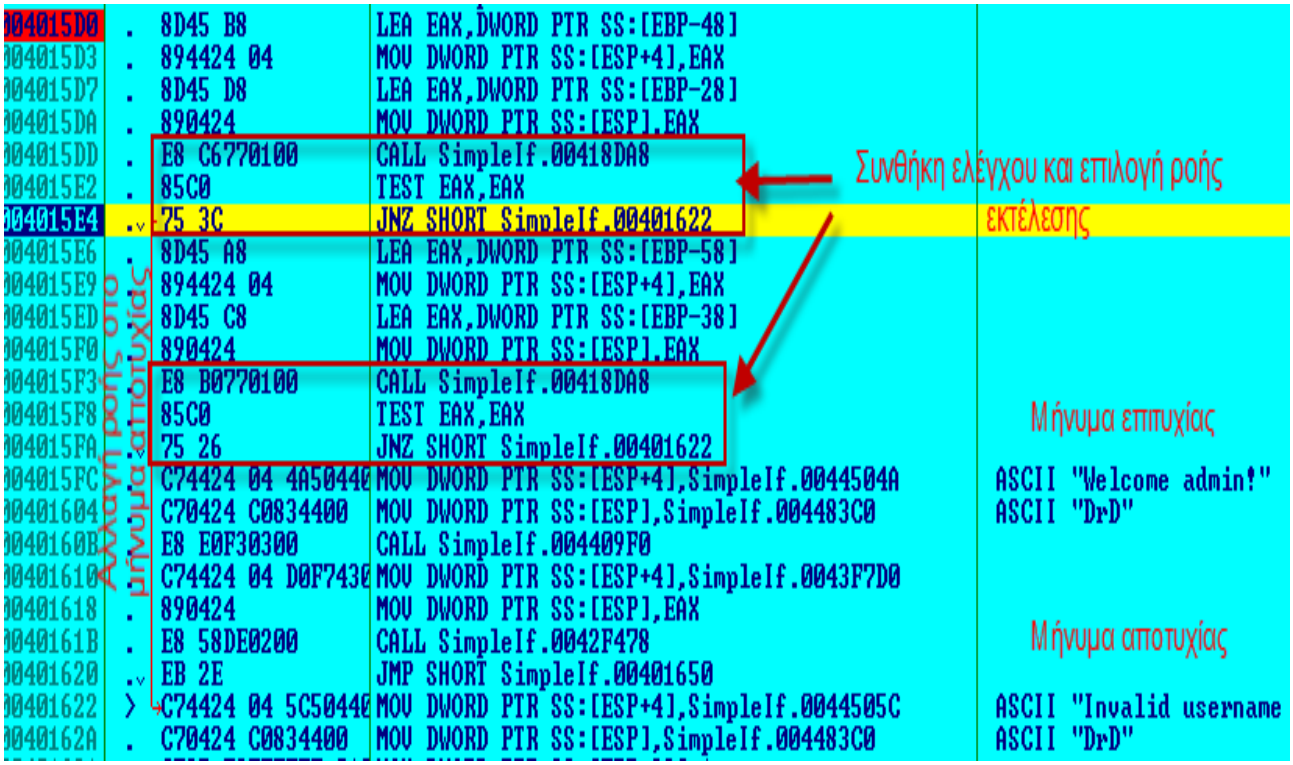
00401599 . C74424 04 2F504400 MOV DWORD PTR SS:[ESP+4],SimpleIf.0044502F ; ASCII "Please enter your password"

004015CB . E8 90070400 CALL SimpleIf.00441D60

ενώ μετά την εισαγωγή του κωδικού από τον χρήστη η εκτέλεση συνεχίζεται και παγώνει πάλι στην εντολή

004015D0 . 8D45 B8 LEA EAX,DWORD PTR SS:[EBP-48] .

Τέλος, υπάρχει ένα τμήμα κώδικα στο οποίο διακρίνουμε κάποιες μορφές λογικού ελέγχου και αποφάσεις για την επιλογή της σωστής ή της λανθασμένης ροής εκτέλεσης το οποίο παρουσιάζεται στην Εικόνα 5.4.



Εικόνα 5.4: Επιλογή της λανθασμένης ροής εκτέλεσης

Σε αυτό το σημείο πρέπει να γίνει κατανοητό πως η απόφαση θα παρθεί ανάλογα με τις συνθήκες στην υπορουτίνα «SimpleIf.00418DA8» που θα καθορίσουν την τιμή του καταχωρητή EAX. Η εντολή «TEST EAX, EAX» ουσιαστικά θα επιχειρήσει να εκτελέσει την λογική πράξη «AND» μεταξύ της τιμής του EAX και τον εαυτό της. Σαν αποτέλεσμα αν η

τιμή του καταχωρητή είναι 0 θα αλλάξει την τιμή του καταχωρητή ZF στην τιμή 1 αλλιώς θα την αλλάξει σε 0. Έτσι στην συνέχεια αν ο ZF έχει την τιμή 0 η εντολή «JNZ SHORT SimpleIf.00401622» θα εκτελεστεί με αποτέλεσμα να μεταφερθεί η ροή εκτέλεσης στην υπορουτίνα που περιέχει το μήνυμα αποτυχίας. Σε περίπτωση που είναι 1 θα συνεχιστεί η εκτέλεση μέχρι το σημείο που θα γίνει η κλήση της υπορουτίνας «SimpleIf.00418DA8» οπότε και θα αρχίσει ένας νέος έλεγχος όπως και προηγουμένως. Έτσι έχει σχεδόν ολοκληρωθεί η ανάλυση και έχουν εντοπιστεί οι δομές και οι λογικοί έλεγχοι που καθορίζουν τη ροή της εκτέλεσης της εφαρμογής και ο μηχανικός λογισμικού μπορεί να προχωρήσει στην επιδιόρθωση της εφαρμογής.

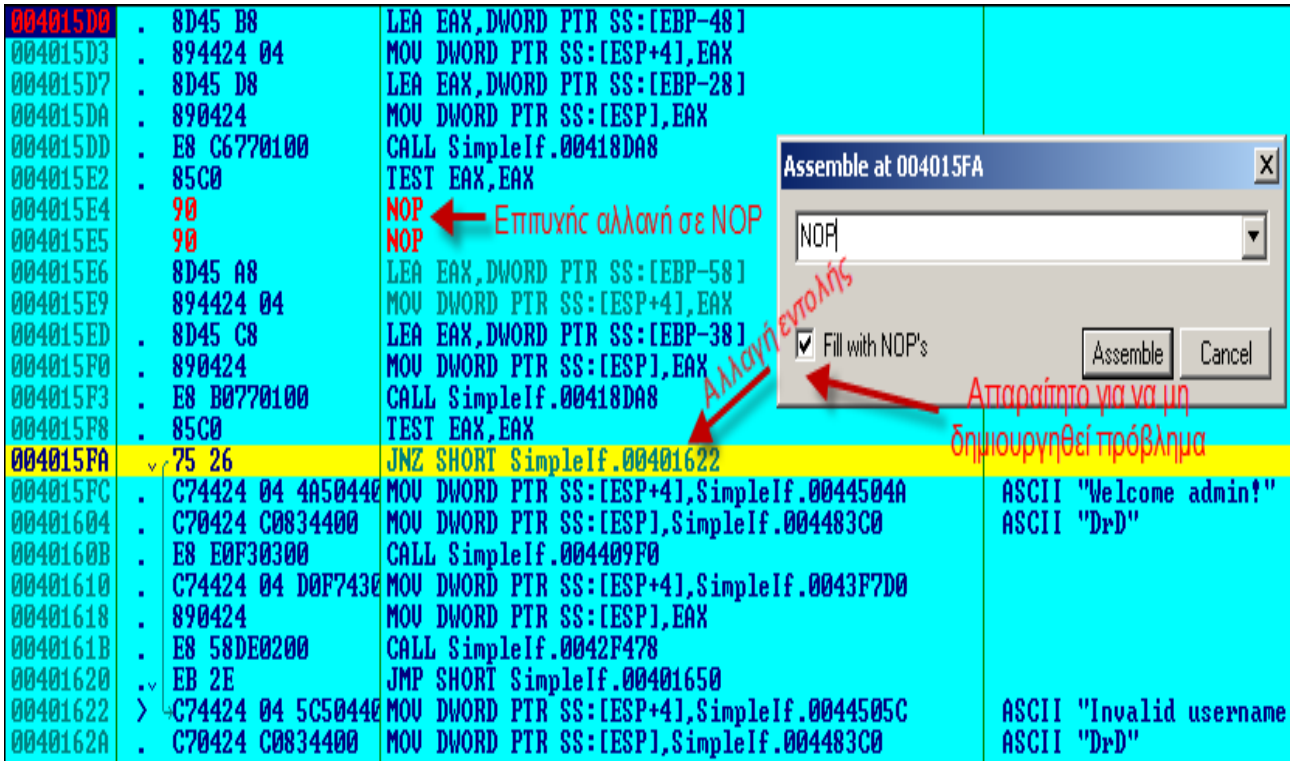
5.2.2 Επιδιόρθωση της εφαρμογής

ΒΗΜΑ 1ο

Ο στόχος της επιτυχούς εκτέλεσης της εφαρμογής με λάθος αναγνωριστικό χρήστη και κωδικό είναι πλέον εφικτός με πολλές πιθανές τροποποιήσεις στον κώδικα. Μια πιθανότητα θα ήταν να γίνει αλλαγή του κώδικα «JNZ SHORT 00401622» στις διευθύνσεις 004015E4 και 004015FA με την εντολή NOP (No Operation) η οποία δεν κάνει τίποτα απολύτως. Δηλαδή ακυρώνουμε την εκτέλεση της μη επιθυμητής εντολής και γεμίζουμε τον χώρο με κενές εντολές. Αυτό επιτυγχάνεται εύκολα στον OllyDbg κάνοντας διπλό κλικ πάνω στην εντολή προς τροποποίηση. Ακόμα είναι σημαντικό να συμπληρωθούν τόσα NOP ώστε να καλυφθεί οποιαδήποτε διαφορά μεγέθους ανάμεσα στην εντολή που αντικαθιστούμε και την καινούρια εντολή. Μια εντολή NOP αντιστοιχεί στον τελεστή 90_{16} ενώ οι εντολές που αφαιρέθηκαν μπορεί να έχουν περισσότερους τελεστές (στη συγκεκριμένη περίπτωση δύο, επομένως θα χρειαστούν δύο NOP). Σε περίπτωση που αυτό δε συμβεί οι αλλαγές μπορεί να είναι απρόβλεπτες και η εφαρμογή να μη λειτουργεί ορθά. Ο OllyDbg δίνει τη δυνατότητα αυτόματης συμπλήρωσης με NOP ώστε να αποφεύγονται τέτοια προβλήματα. Στην Εικόνα 5.5 παρουσιάζονται οι αλλαγές στον κώδικα ώστε να επιτευχθεί η αλλαγή στην ροή εκτέλεσης που εμείς επιθυμούμε.

Σε αυτό το σημείο να συμπληρώσουμε πως αυτός δεν είναι ο μόνος τρόπος επιδιόρθωσης του κώδικα καθώς αυτό αφήνεται στην εμπειρία, την ευρηματικότητα και τη

φαντασία του κάθε μηχανικού λογισμικού ώστε να επιλέξει τον καλύτερο δυνατό τρόπο. Έτσι για παράδειγμα μια άλλη πιθανή αλλαγή θα ήταν να αντικατασταθεί η κλήση «CALL 00418DA8» στις δύο διευθύνσεις 004015DA και 004015F3 με την εντολή MOV EAX, 0 ώστε να θέσουμε την τιμή του EAX σε 0 πριν τον έλεγχο της από την TEST EAX, EAX με το ίδιο τελικό αποτέλεσμα. Γενικά όμως καθώς η εκτέλεση μιας ολόκληρης υπορουτίνας είναι πιθανό να είναι εντελώς απαραίτητη για την ομαλή λειτουργία μιας εφαρμογής καλό είναι να επιλέγεται η μέθοδος με την μικρότερη επιρροή στην εφαρμογή.

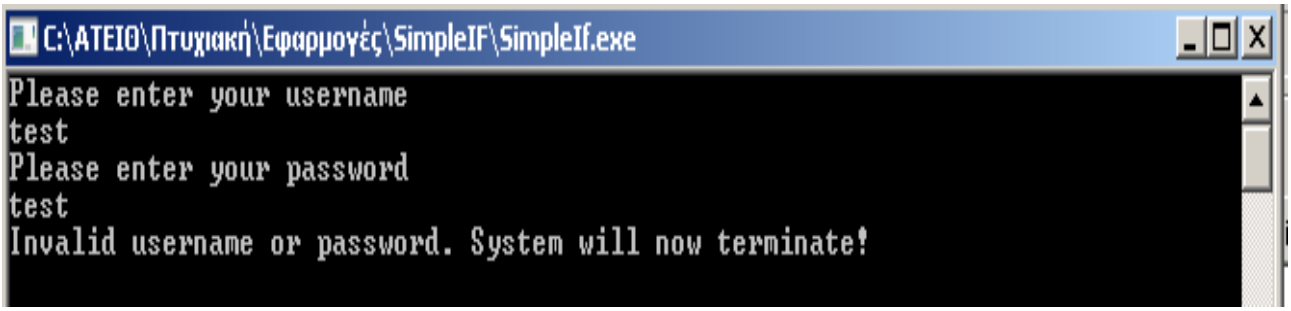


Εικόνα 5.5: Επιδιόρθωση του κώδικα

ΒΗΜΑ 2ο

Εφόσον έχουν βρεθεί και ολοκληρωθεί όλες οι αλλαγές που θα οδηγήσουν στον επιθυμητό αποτέλεσμα, το επόμενο βήμα είναι η μόνιμη αποθήκευση των αλλαγών σε ένα καινούριο αρχείο. Αυτό γίνεται εύκολα στον OllyDbg κάνοντας δεξί κλικ στο παράθυρο του Disassembler και επιλέγοντας «Copy to executable» → «All modifications». Στη συνέχεια στο παράθυρο διαλόγου ρωτάει ο OllyDbg αν θέλουμε να αποθηκεύσει όλες τις αλλαγές και δίνει την επιλογή «Copy All». Τέλος εμφανίζει το παράθυρο «DUMP» το οποίο περιέχει

τον κώδικα του εκτελέσιμου με όλες τις τροποποιήσεις στο οποίο κάνοντας δεξί κλικ δίνεται η επιλογή αποθήκευσης του εκτελέσιμου σε ένα νέο αρχείο (Backup → Save Data to File). Το νέο αρχείο πλέον εκτελείται και δουλεύει έχοντας αφαιρεμένη την ασφάλεια του. Στην Εικόνα 5.6 φαίνεται το αποτέλεσμα της εκτέλεσης του αρχικού εκτελέσιμου και στην Εικόνα 5.7 το αποτέλεσμα της εκτέλεσης του τροποποιημένου εκτελέσιμου.



Εικόνα 5.6: Εκτέλεση της αρχικής εφαρμογής με λανθασμένα στοιχεία

Εικόνα 5.7: Εκτέλεση της τροποποιημένης εφαρμογής με λανθασμένα στοιχεία

5.2.3 Εναλλακτική εύρεση σημείων ενδιαφέροντος της εφαρμογής (αναζήτηση συμβολοσειρών).

Σίγουρα μπορεί κανείς να διαπιστώσει πως το μεγαλύτερο και ποιο σημαντικό κομμάτι της ανάλυσης της εφαρμογής είναι η κατανόηση της συμπεριφοράς του προγράμματος και η αναγνώριση των τμημάτων που καθορίζουν αυτή τη συμπεριφορά. Έτσι καθώς εκτελούνταν ο κώδικας γραμμή προς γραμμή αποκαλύπτονταν πληροφορίες και μηνύματα που μπορεί να δει κάποιος αν απλά εκτελέσει την εφαρμογή τα οποία με τη σειρά τους διευκόλυναν την αναγνώριση αυτών των κρίσιμων τμημάτων. Ο τρόπος αυτός όμως είναι πολύ χρονοβόρος καθώς η εφαρμογή που είδαμε θα μπορούσε να είναι μόνο ένα τμήμα μιας αρκετά μεγαλύτερης εφαρμογής που χειρίζεται την είσοδο του χρήστη.

Εύλογα λοιπόν αναρωτιέται κανείς αν υπάρχει κάποιος πιο εύκολος τρόπος να εντοπιστεί αυτό το σημείο.

Η δυνατότητα αυτή παρέχεται από τον OllyDbg καθώς αναγνωρίζει όλες τις συμβολοσειρές που υπάρχουν στην εφαρμογή και επιτρέπει στον χρήστη να αναζητήσει αυτές τις συμβολοσειρές. Στο παράδειγμα μας είδαμε ένα μήνυμα αποτυχίας και ένα μήνυμα επιτυχίας τα οποία αποτελούν συμβολοσειρές με βάση τις οποίες θα αναλυθεί η εφαρμογή. Όπως και πριν τα πρώτα δύο βήματα της ενότητας 5.2.1 θα πρέπει να εφαρμοστούν και η αλλαγή θα παρουσιαστεί στο 3ο βήμα.

ΒΗΜΑ 1ο (βλ. ενότητα 5.2.1)

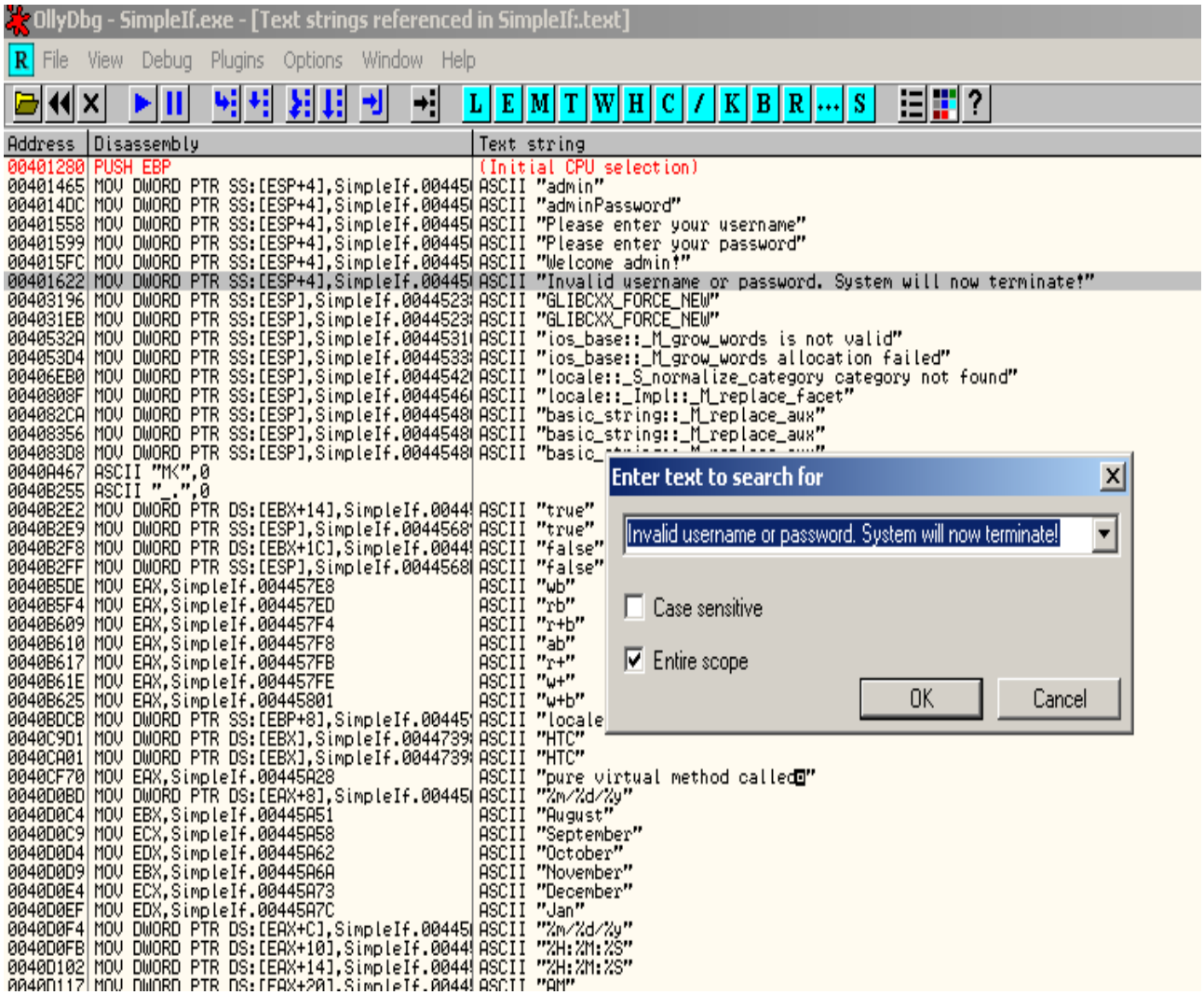
ΒΗΜΑ 2ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 3ο

Μετά την εκτέλεση του κώδικα ως το σημείο εισόδου του κώδικα της εφαρμογής SimpleIF και κάνοντας δεξί κλικ στο παράθυρο του disassembler του OllyDbg δίνεται η επιλογή για αναζήτηση όλων των αναφερόμενων συμβολοσειρών κειμένου (Search For → All referenced text strings) και ο OllyDbg παρουσιάζει ένα παράθυρο με συμβολοσειρές και τις διευθύνσεις στις οποίες βρέθηκαν. Ακόμα κάνοντας δεξί κλικ στο καινούριο παράθυρο μπορεί κάποιος να πραγματοποιήσει αναζητήσεις μέσα σε αυτά (επιλογή «search for text»). Σημαντική παρατήρηση αποτελεί το γεγονός πως η αναζήτηση γίνεται από πάνω προς τα κάτω πάντα και μόνο στο τμήμα που φαίνεται στο παράθυρο εκτός αν στο παράθυρο διαλόγου της αναζήτησης επιλέξει κανείς να γίνει αναζήτηση σε όλο το εύρος (Entire Scope). Καλό είναι επίσης κατά τις αναζητήσεις να επιλέγεται ο μη διαχωρισμός κεφαλαίων-μικρών (απεπιλογή του «case sensitive») όπως φαίνεται στην

Εικόνα 5.8 που αναζητούμε το μήνυμα αποτυχίας «Invalid username or password. System will now terminate!». Στη συνέχεια μπορούμε να ακολουθήσουμε την

συμβολοσειρά στην διεύθυνση στην οποία εμφανίζεται (00401622) και να ορίσουμε ένα σημείο διακοπής λογισμικού. Έπειτα ακολουθώντας αντίστροφα τις εντολές μπορούμε να αναγνωρίσουμε τις δομές οι οποίες οδήγησαν στην εκτέλεση του κώδικα και τις αλλαγές ροής που κατευθύνονται στο μήνυμα αποτυχίας. Τέλος η επιδιόρθωση του κώδικα γίνεται όπως αναφέρθηκε στην ενότητα 5.2.2 αλλά ο χρόνος ανάλυσης της εφαρμογής έχει μειωθεί σημαντικά.



Εικόνα 5.8: Αναζήτηση μηνύματος αποτυχίας

5.3 Προχωρημένη επιδιόρθωση κώδικα (Advanced Patching).

Σε πολλές εφαρμογές η επιδιόρθωση του κώδικα δεν είναι τόσο απλή καθώς μπορεί να υπάρχει συνεχής πιστοποίηση των στοιχείων του χρήστη ανά τακτά χρονικά διαστήματα με αποτέλεσμα μια απλή αλλαγή ροής να μην είναι αρκετή ώστε να παρακάμψει την ασφάλεια της εφαρμογής εκτός και αν κάποιος σπαταλήσει αρκετό χρόνο ώστε να ανακαλύψει και επιδιορθώσει όλες τις πιθανές διαδρομές που μπορούν να οδηγήσουν στο ανεπιθύμητο αποτέλεσμα. Ακόμα όμως και τότε μια μικρή αλλαγή της εφαρμογής σε κάποια αναβάθμιση με την προσθήκη καινούριων δυνατοτήτων μπορεί να οδηγήσει στη δημιουργία νέων πιθανών διαδρομών που οδηγούν σε ανεπιθύμητα αποτελέσματα.

Σε τέτοιες περιπτώσεις ο μηχανικός λογισμικού πρέπει να εμβαθύνει περισσότερο και να ανακαλύψει το σημείο στο οποίο γίνεται η πιστοποίηση και να τοποθετήσει τον επιδιορθωμένο κώδικα με τέτοιο τρόπο ώστε να επιτύχει έμμεσα την αλλαγή ροής που επιθυμεί.

Αυτό θα φανεί στην προσπάθεια να επιτευχθεί αλλαγή στην ροή εκτέλεσης στο εκτελέσιμο αρχείο της Εφαρμογή SimpleValidate.cpp (βλ. Παράρτημα Α).

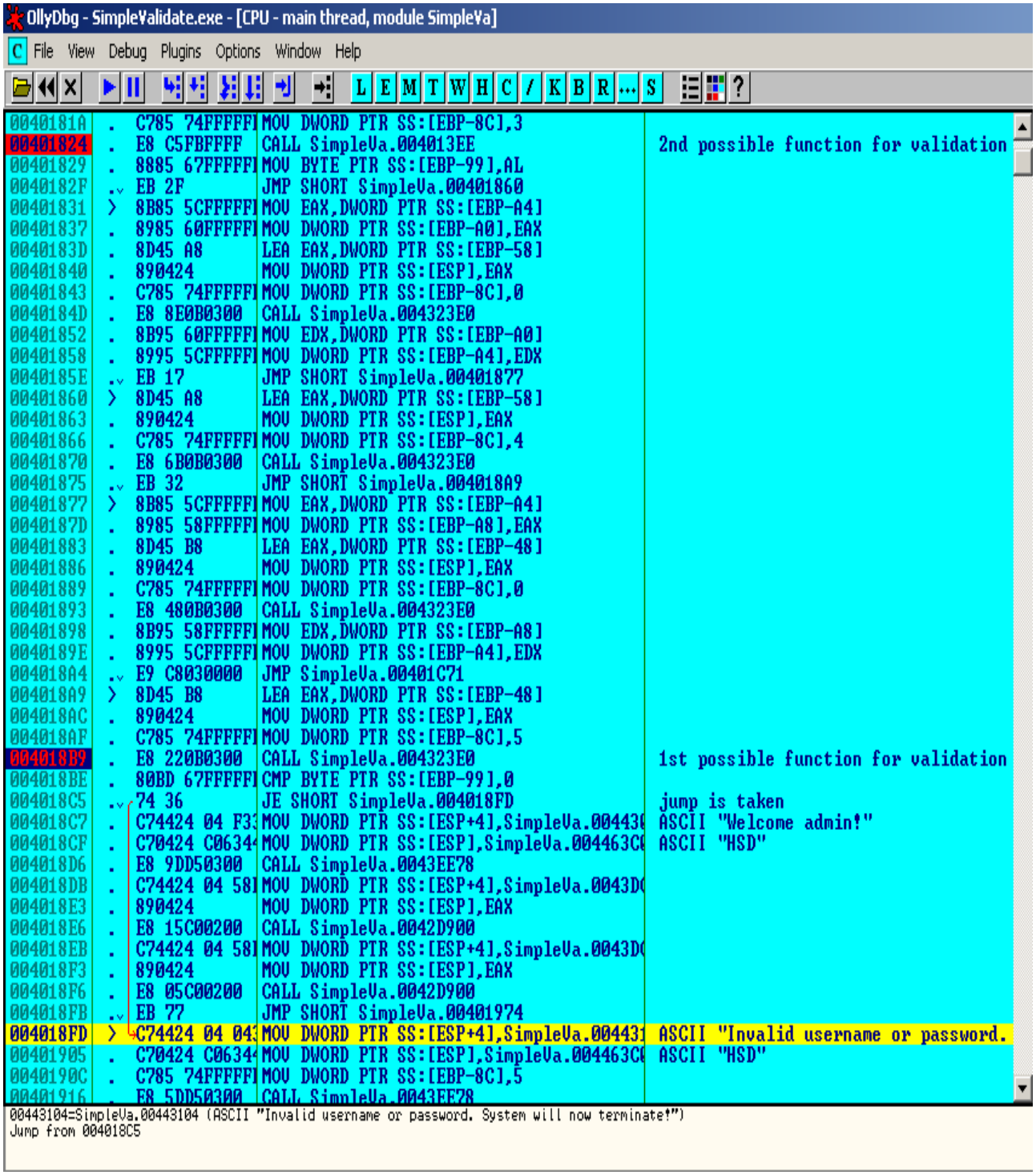
5.3.1 Ανάλυση της εφαρμογής

ΒΗΜΑ 1ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 2ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 3ο

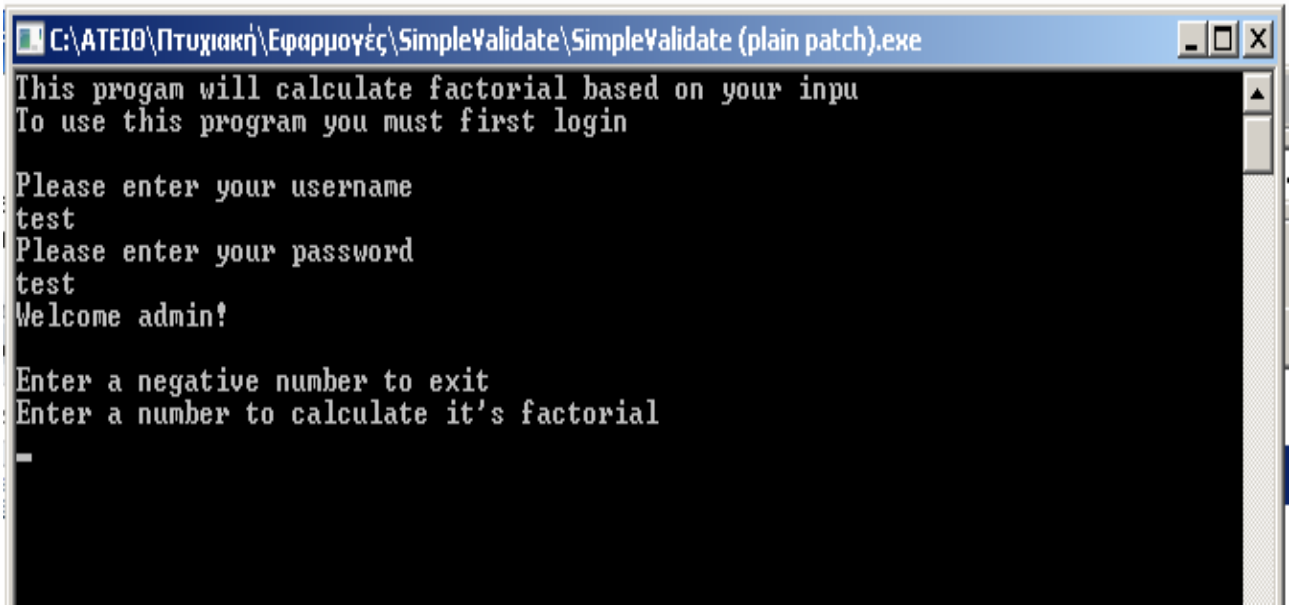
Στην συγκεκριμένη εφαρμογή παρατηρείται πως εμφανίζονται κάποια βασικά μηνύματα για την εισαγωγή του αναγνωριστικού χρήστη (username) και του κωδικού πρόσβασης (password) του. Τέλος όταν αυτά εισαχθούν λανθασμένα εμφανίζεται ένα μήνυμα λάθους και μια προειδοποίηση τερματισμού της εφαρμογής.



Εικόνα 5.9: Μήνυμα αποτυχίας στην εφαρμογή SimpleValidate.cpp

Χρησιμοποιώντας την τεχνική αναζήτησης συμβολοσειρών κειμένου όπως παρουσιάστηκε στην ενότητα 5.2.3 αναζητείται το μήνυμα αποτυχίας. Όπως παρουσιάζεται στην Εικόνα 5.9 ο OllyDbg εμφανίζει τη διεύθυνση 004018FD που βρίσκεται το μήνυμα αποτυχίας ενώ έχει ήδη μεταφράσει το μήνυμα αυτό σε χαρακτήρες ASCII. Επίσης ενημερώνει ότι η αλλαγή ροής έγινε από την διεύθυνση 004018C5. Αμέσως

προηγουμένως υπάρχει μια εντολή σύγκρισης που είναι προφανές πλέον πως θα καθορίσει την επιτυχή ή μη αλλαγή ροής εκτέλεσης της εφαρμογής. Εύκολα θα υπέθετε κάποιος πως για να επιτευχθεί η ζητούμενη αλλαγή επαρκεί μια αλλαγή της εντολής «JE SHORT 004018FD» σε «NOP» ώστε να αποφευχθεί η εμφάνιση του μηνύματος αποτυχίας και να συνεχίσει το πρόγραμμα την σωστή του ροή εκτέλεσης. Όντως συνεχίζοντας την εκτέλεση μετά την επιδιόρθωση παρατηρεί κανείς στην Εικόνα 5.10 ότι η εφαρμογή εμφανίζει το μήνυμα επιτυχίας και ζητάει από τον χρήστη να εισάγει έναν αριθμό για να υπολογίσει το παραγοντικό του, μόλις όμως ο χρήστης εισάγει έναν αριθμό η εφαρμογή κλείνει απροσδόκητα.



Εικόνα 5.10: Απλή επιδιόρθωση του κώδικα της εφαρμογής (SimpleValidate)

Επομένως μπορεί εύκολα να συμπεράνει κανείς πως η απλή επιδιόρθωση του κώδικα σε αυτήν την περίπτωση δεν ήταν αρκετή για να οδηγήσει στην επιτυχή εκτέλεση της εφαρμογής και άρα κάποια υπορουτίνα που κλήθηκε προηγουμένως είναι υπεύθυνη για την πιστοποίηση του χρήστη. Επομένως ο μηχανικός λογισμικού πρέπει να μελετήσει όλες τις προηγούμενες κλήσεις κατά σειρά (χρησιμοποιώντας step into). Πηγαίνοντας λοιπόν αντίστροφα παρατηρεί κανείς ότι εμφανίζεται πολύ συχνά η κλήση **«CALL SimpleVa.004323E0»** με πρώτη στην διεύθυνση 00401870 και στη συνέχεια στην 004018B9 ενώ προηγουμένως στην διεύθυνση 00401824 υπάρχει η κλήση **«CALL SimpleVa.004013EE»**. Και οι δύο αυτές υπορουτίνες έχουν αρκετές πιθανότητες να καθορίζουν την τιμή στη διεύθυνση **«BYTE PTR SS:[EBP-99]»** που αποτελεί το πρώτο όρισμα της κρίσιμης σύγκρισης για την αλλαγή ροής καθώς σε περίπτωση που η τιμή της

είναι 0 τότε επιτυγχάνεται η αλλαγή ροής και εμφανίζεται το μήνυμα αποτυχίας. Εισάγοντας ένα σημείο διακοπής λογισμικού στην γραμμή κώδικα

*004018BE . 80BD 67FFFFFF>**CMP BYTE PTR SS:[EBP-99],0***

και εκτελώντας την εφαρμογή μέχρι αυτό το σημείο μπορεί να δει κάποιος στον OllyDbg όπως φαίνεται στην Εικόνα 5.11 την διεύθυνση της καθοριστικής τιμής για την αλλαγή ροής (0028FEAF).

Ακόμα μπορεί να δει πως η τιμή της ισούται με το μηδέν μετά την εισαγωγή λανθασμένων στοιχείων. Αν θέλει κανείς μπορεί να παρακολουθήσει αυτήν την τιμή από την αρχή της εκτέλεσης της εφαρμογής κάνοντας δεξί κλικ στο παράθυρο που δείχνει το εκτελέσιμο αρχείο (Dump Window) και επιλέγοντας να μεταφερθεί στην «έκφραση» (expression) δηλαδή στην διεύθυνση 0028FEAF (Go To → Expression) ώστε να βλέπει πότε θα γίνει η κρίσιμη μεταβολή στην τιμή μηδέν. Έτσι μπορεί να καταλήξει πως η κλήση «CALL SimpleVa.004313EE» στην διεύθυνση 00401824 καθορίζει την κρίσιμη τιμή. Εκτός από αυτήν την ένδειξη είναι δυνατό, εκτελώντας τον κώδικα με την απλή επιδιόρθωση που έγινε αρχικά, να εντοπιστεί ένα ακόμα κρίσιμο σημείο στη ροή εκτέλεσης. Πρόκειται για το σημείο στο οποίο τερματίζει απρόσμενα η εφαρμογή.

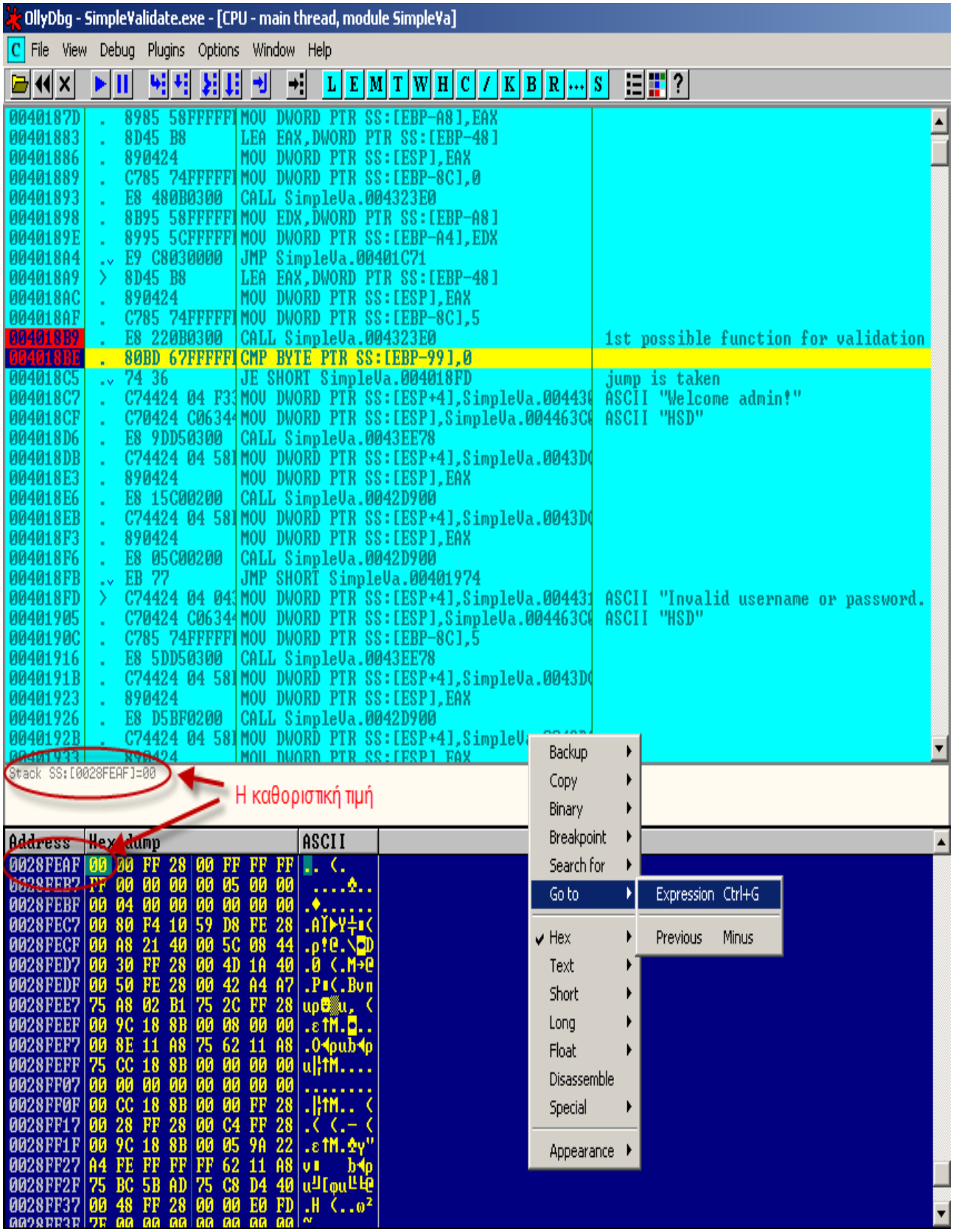
Στην Εικόνα 5.12 παρουσιάζεται ένα σημείο παρόμοιο με τον κρίσιμο κώδικα πριν το μήνυμα αποτυχίας και μια καθοριστική αλλαγή ροής που τερματίζει την εφαρμογή. Επιδιορθώνοντας τον κώδικα που ευθύνεται για την αλλαγή ροής αυτή και αλλάζοντας την εντολή

00401B39 /74 39 JE SHORT SimpleVa.00401B74 ; jump is not taken

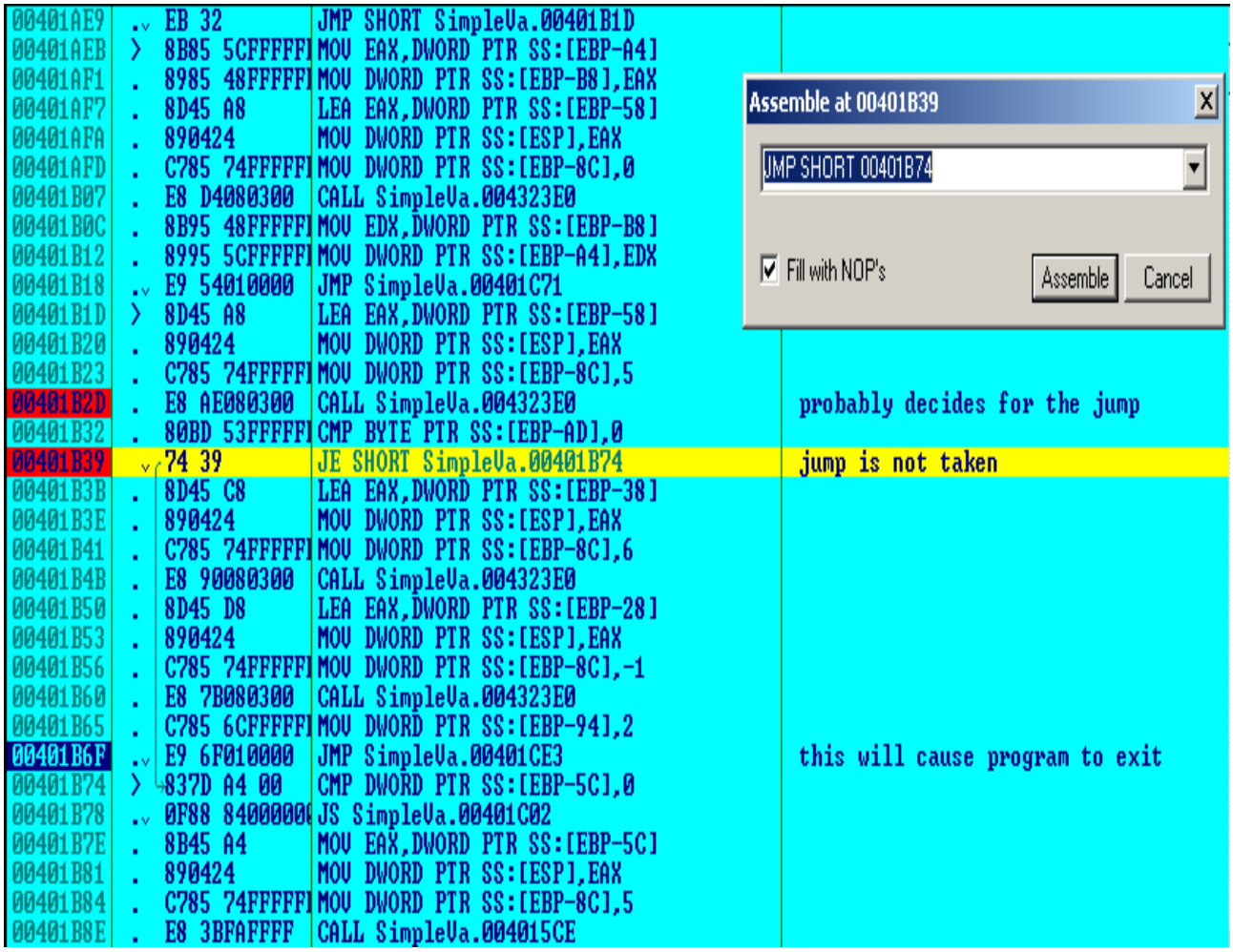
στην

00401B39 /EB 39 JMP SHORT SimpleVa.00401B74 ; jump is always taken

αποφεύγεται ο απρόσμενος τερματισμός της εφαρμογής και εκτελείται κανονικά η εφαρμογή. Συμπεραίνει κανείς λοιπόν ότι πρόκειται για έναν δεύτερο «κρυμμένο» έλεγχο της ταυτότητας του χρήστη. Επομένως με τον συνδυασμό δύο απλών επιδιορθώσεων έχει παρακαμφθεί η ασφάλεια της εφαρμογής. Το πρόβλημα όμως δεν έχει λυθεί οριστικά καθώς ο μηχανικός δεν γνωρίζει πόσες φορές μπορεί να ελέγχεται η ταυτότητα του χρήστη



Εικόνα 5.11: Η καθοριστική τιμή για την αλλαγή ροής



Εικόνα 5.12: Σημείο απρόσμενης διακοπής μετά την απλή επιδιόρθωση

όπως προαναφέραμε. Όμως αν συνυπολογισθεί το γεγονός πως πάλι πριν την εκτέλεση της υπορουτίνας «SimpleVa.004323E0» προηγείται η εκτέλεση της υπορουτίνας «SimpleVa.004013EE» όπως γίνεται και στο Τμήμα Κώδικα 5.1.

```
00401A3B . E8 AEF9FFFF CALL SimpleVa.004013EE
00401A40 . 34 01 XOR AL,1
00401A42 . 8885 53FFFFFF MOV BYTE PTR SS:[EBP-AD],AL
00401A48 . E9 87000000 JMP SimpleVa.00401AD4
```

Τμήμα Κώδικα 5.1: Simple Validate

το οποίο εκτελείται πριν το σημείο που δείχνει η Εικόνα 5.12 και αμέσως μετά ορίζεται η τιμή της διεύθυνσης **«BYTE PTR SS:[EBP-AD]»** η οποία περιέχει την καθοριστική τιμή για την εκτέλεση της αλλαγής ροής στο Τμήμα Κώδικα 5.2

```
00401B2D . E8 AE080300 CALL SimpleVa.004323E0 ; probably decides for the jump
```

```
00401B32  .  80BD 53FFFFFF>CMP BYTE PTR SS:[EBP-AD],0
00401B39     74 39        JE SHORT SimpleVa.00401B74          ;  jump is not taken
```

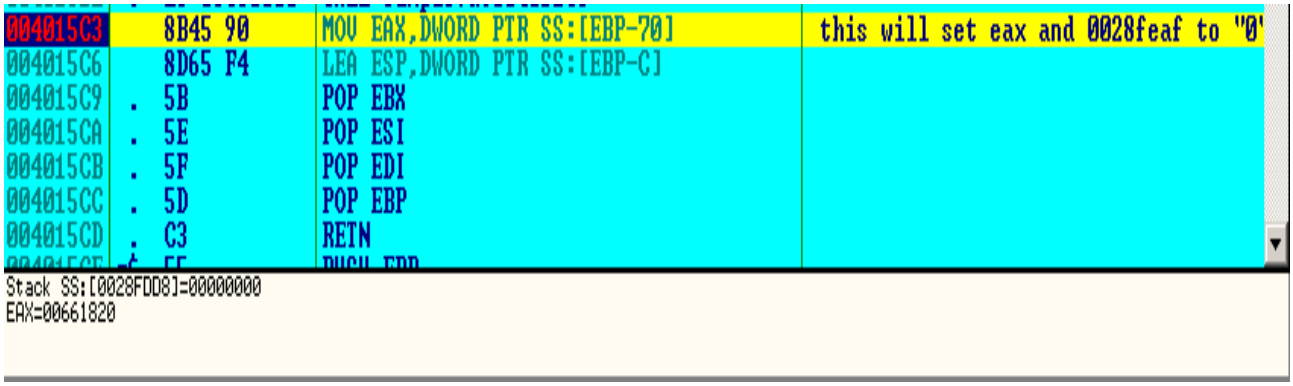
Τμήμα Κώδικα 5.2: Simple Validate

τότε συμπεραίνει κανείς πως η υπορουτίνα που είναι υπεύθυνη για την πιστοποίηση του χρήστη είναι η **«SimpleVa.004013EE»**.

Άρα ο σωστός τρόπος λύσης είναι να βρεθεί το σημείο μέσα σε αυτήν που καθορίζει την τιμή η οποία ελέγχεται στη συνέχεια για την ισότητα της με το μηδέν πριν τις αλλαγές ροής. Ακόμα συνυπολογίζοντας το γεγονός πως και στις δύο περιπτώσεις υπάρχει ένα τμήμα κώδικα το οποίο ορίζει την τιμή στις διευθύνσεις **«BYTE PTR SS:[EBP-99]»** και **«BYTE PTR SS:[EBP-AD]» SS:[EBP-AD]»** με βάση τον καταχωρητή EAX (ή το τμήμα AL), είναι πολύ πιθανό απλά να χρειάζεται να καθοριστεί η τιμή του EAX στο τέλος της εκτέλεσης της «SimpleVa.004013EE».

ΒΗΜΑ 4ο

Σε αυτό το σημείο πρέπει να γίνει εκτενέστερη ανάλυση της υπορουτίνας **«SimpleVa.004013EE»** και άρα πρέπει να εισαχθεί ένα σημείο διακοπής λογισμικού στην πρώτη της εντολή. Στη συνέχεια εκτελώντας τις εντολές της με τη σειρά φτάνουμε στο καταληκτικό τμήμα κώδικα όπως φαίνεται στην Εικόνα 5.13 όπου φαίνεται ξεκάθαρα πως



Εικόνα 5.13:Το σημείο που καθορίζει την τιμή του EAX

στη διεύθυνση 004015C3 καθορίζεται η τιμή του καταχωρητή EAX ενώ στην συνέχεια δεν επηρεάζεται από κάποια εντολή μέχρι την ολοκλήρωση της υπορουτίνας και την επιστροφή στην αρχική ροή της εφαρμογής.

5.3.2 Επιδιόρθωση της εφαρμογής

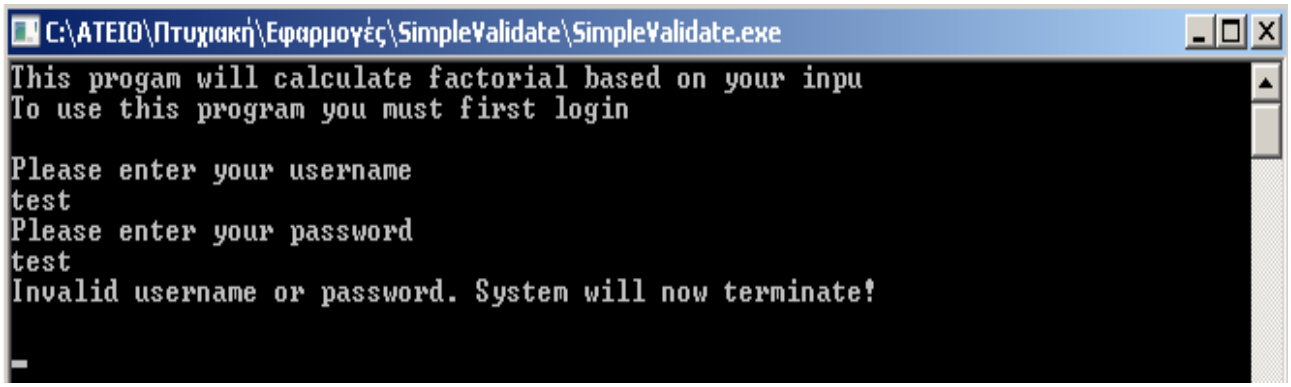
ΒΗΜΑ 1ο

Έχοντας ανακαλύψει το σημείο που γίνεται ο ορισμός της τιμής του καταχωρητή EAX θα μπορούσε κανείς να σκεφτεί πως μια απλή αλλαγή της τιμής «DWORD PTR SS:[EBP-70]|» στην τιμή 1 αντί για 0 ώστε να πάρει ο EAX την τιμή 1 θα είναι η σωστή λύση όμως αυτό δεν είναι εφικτό καθώς για να οριστεί η εντολή **«MOV EAX, 1»** (με αντίστοιχους κωδικούς λειτουργίας: B8 01000000) απαιτούνται 5 bytes ενώ η ήδη υπάρχουσα εντολή καταλαμβάνει χώρο τριών byte (συνυπολογίζοντας τον κωδικό λειτουργίας 90 που αντιστοιχεί στην εντολή NOP) και επομένως για να επιτευχθεί η αλλαγή θα χρειαστεί να αλλαχθεί και η επόμενη εντολή πράγμα το οποίο ενδεχομένως να οδηγήσει σε προβληματική συμπεριφορά. Πράγματι αν το επιχειρήσει κανείς θα δει πως η εφαρμογή σταματά να λειτουργεί.

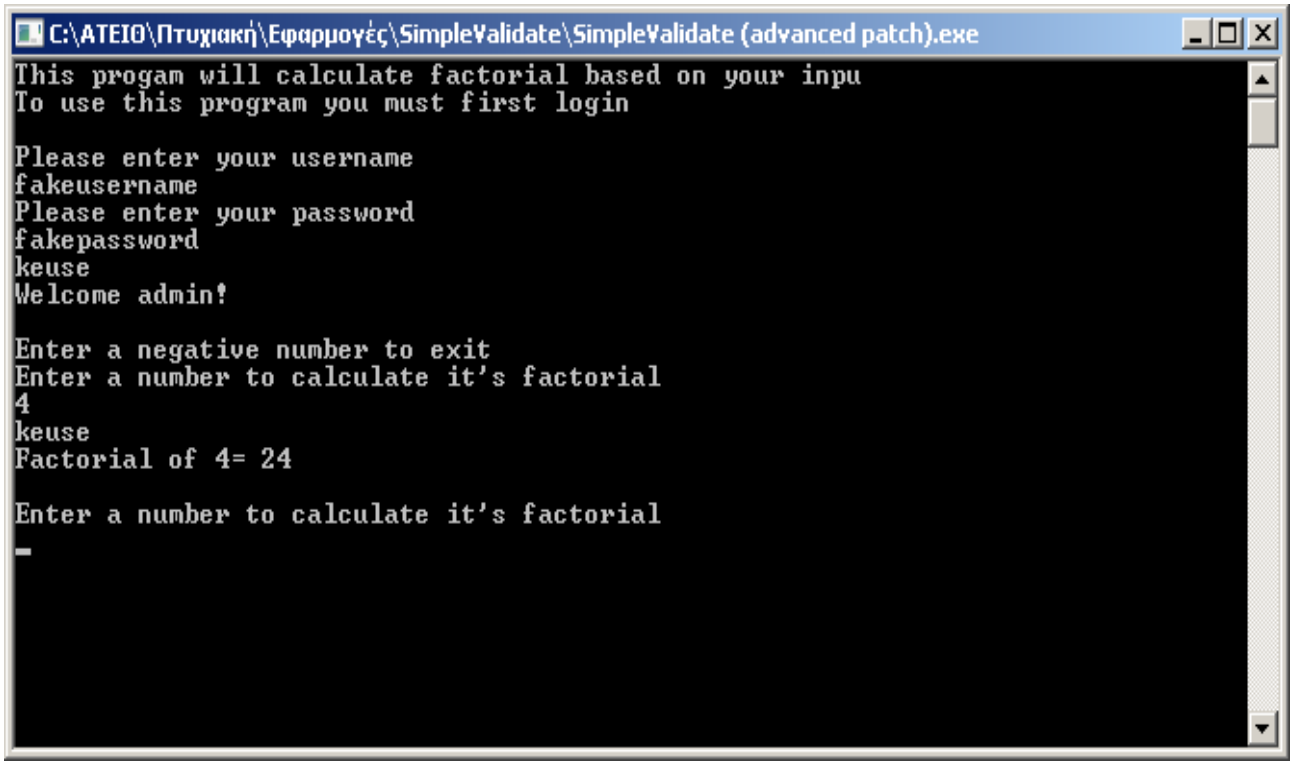
Τελικά η λύση δίνεται αλλάζοντας την εντολή «MOV EAX,DWORD PTR SS:[EBP-70]» σε «XOR EAX, EAX» η οποία εντολή πάντα θα μηδενίζει τον καταχωρητή EAX και καταλαμβάνει μόνο 2 bytes ενώ στη συνέχεια, στη θέση της εντολής NOP που προσθέτει ο OllyDbg μπορεί να γίνει αντικατάσταση με την εντολή «INC EAX» και έτσι η τιμή του EAX γίνεται 1. Η εντολή αυτή καταλαμβάνει μόνο 1 byte. Πράγματι με την αλλαγή αυτή η εφαρμογή εκτελείται κανονικά χωρίς προβλήματα.

ΒΗΜΑ 2ο

Εφόσον έχουν ολοκληρωθεί οι αλλαγές πρέπει αυτές να αποθηκευτούν σε ένα νέο αντίγραφο του εκτελέσιμου με τη μέθοδο που παρουσιάστηκε στην ενότητα 5.2.2. Στην Εικόνα 5.14 φαίνεται το αποτέλεσμα της εκτέλεσης του αρχικού εκτελέσιμου και στην Εικόνα 5.15 το αποτέλεσμα της εκτέλεσης του τροποποιημένου εκτελέσιμου.



Εικόνα 5.14: Εκτέλεση της αρχικής εφαρμογής με λανθασμένα στοιχεία



Εικόνα 5.15: Εκτέλεση της τροποποιημένης εφαρμογής με λανθασμένα στοιχεία

5.4 Εύρεση στοιχείων χρήστη και γεννήτριες κλειδιών (Keygen).

Επειδή πολλές φορές η επιδιόρθωση του κώδικα μπορεί να έχει απροσδόκητα αποτελέσματα όπως αστοχίες ή σφάλματα λογισμικού, η παράκαμψη της ασφάλειας του λογισμικού απαιτεί την αντιστροφή του τμήματος της εφαρμογής που χειρίζεται την ταυτοποίηση του χρήστη. Αυτό μπορεί να γίνει με στόχο την παραγωγή ενός μοναδικού

κλειδιού το οποίο είναι λειτουργικό ή με στόχο τη δημιουργία μιας εφαρμογής – γεννήτριας κλειδιών τα οποία θεωρούνται από την εφαρμογή αυθεντικά.

Σε αυτό το σημείο θα γίνει μια προσπάθεια να αντιστραφεί ο αλγόριθμος ταυτοποίησης της Εφαρμογή SimpleValidate.cpp (βλ. Παράρτημα Α) σε συνέχεια της ενότητας 5.3.

5.4.1 Αντιστροφή του αλγόριθμου ταυτοποίησης

ΒΗΜΑ 1ο

Όπως παρουσιάστηκε στην ενότητα 5.3.1 η υπορουτίνα **«SimpleVa.004013EE»** είναι υπεύθυνη για την ταυτοποίηση του χρήστη. Επομένως για να βρεθεί ένας αυθεντικός κωδικός θα πρέπει να αντιστραφεί σχεδόν ολόκληρη η λειτουργία της. Εκτελώντας γραμμή προς γραμμή τις εντολές της και έχοντας θέσει ένα καινούριο σημείο διακοπής στην πρώτη της εντολή

```
004013EE   $  55          PUSH EBP
```

παρατηρείται το Τμήμα Κώδικα 5.3

```
00401421   .  E8 C2C60000   CALL SimpleVa.0040DAE8
00401426   .  8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]
00401429   .  890424        MOV DWORD PTR SS:[ESP],EAX
0040142C   .  C745 98 FFFFF>MOV DWORD PTR SS:[EBP-68],-1
00401433   .  E8 985D0100   CALL SimpleVa.004171D0
00401438   .  83F8 09       CMP EAX,9                    ;  if string tested is over 10 chars jump?
0040143B   .  77 0C         JA SHORT SimpleVa.00401449
0040143D   .  C745 90 00000>MOV DWORD PTR SS:[EBP-70],0
00401444   .  E9 6F010000   JMP SimpleVa.004015B8
```

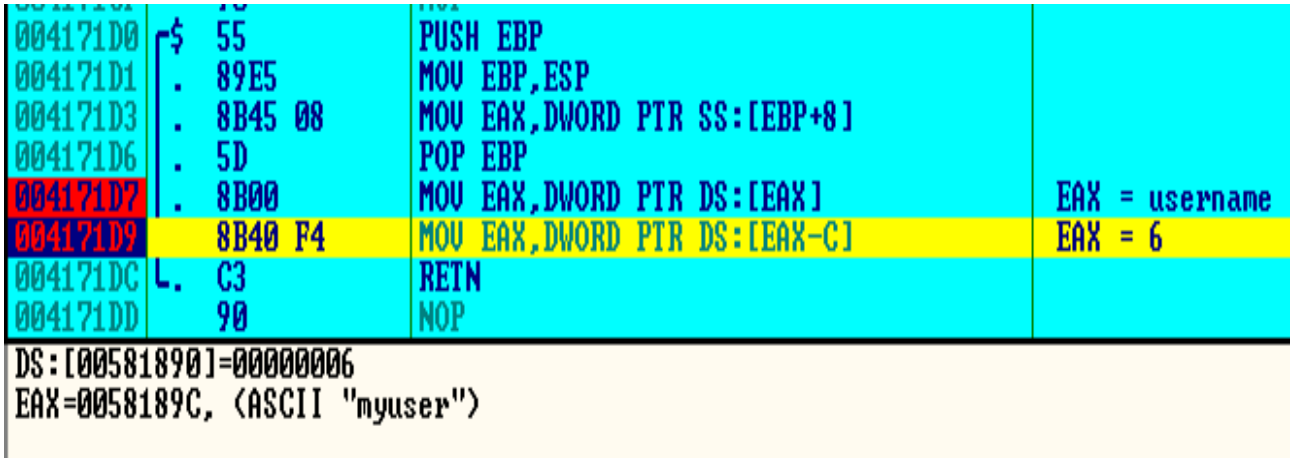
Τμήμα Κώδικα 5.3: Simple Validate

Πιο συγκεκριμένα στην διεύθυνση 00401438 ο καταχωρητής EAX συγκρίνεται με τον αριθμό 9 και σε στη συνέχεια εφόσον η τιμή του είναι μεγαλύτερη πραγματοποιείται μια αλλαγή ροής εκτέλεσης σε κάποιο κοντινό σημείο της υπορουτίνας ενώ αν είναι μικρότερη πραγματοποιείται μια μεγαλύτερη αλλαγή ροής. Αυτό το «άλμα» της διεύθυνσης 00401444

παρατηρεί κανείς εκτελώντας τον κώδικα ότι οδηγεί σε έξοδο από την υπορουτίνα και στη συνέχεια στο μήνυμα αποτυχίας. Πρέπει τώρα να είναι κατανοητό όπως είδαμε και σε προηγούμενες περιπτώσεις πως η τιμή του EAX καθορίζει αν θα πραγματοποιηθεί το «άλμα» και η τιμή του EAX προφανώς καθορίζεται σε μία προηγούμενη κλήση σε κάποια υπορουτίνα. Πρώτη επιλογή και με τις μεγαλύτερες πιθανότητες να επηρεάζει τον καταχωρητή EAX είναι η παρακάτω εντολή

```
00401433   .  E8 985D0100   CALL SimpleVa.004171D0
```

Άρα, θα πρέπει να αναλυθεί εκτενέστερα η υπορουτίνα SimpleVa.004171D0 με «Step Into» κατά την εκτέλεση της. Ένα σημείο διακοπής στη διεύθυνση 004171D0 θα επιστρέψει τον έλεγχο στον χρήστη κατά την εκτέλεση της υπορουτίνας. Ο κώδικας της φαίνεται στην Εικόνα 5.16 και ιδιαίτερο ενδιαφέρον παρουσιάζει η εκτέλεση των εντολών στις διευθύνσεις 004171D7 και 004171D9 όπου μεταφέρεται στον EAX η συμβολοσειρά που εισήχθη ως όνομα χρήστη και η τιμή 6 αντίστοιχα. Άρα, η τιμή του EAX ορίζεται σε αυτήν την υπορουτίνα. Επίσης αποκαλύπεται πως όντως υπάρχει κάποια συσχέτιση του ονόματος χρήστη και της τιμής του EAX.



Εικόνα 5.16: Η υπορουτίνα 004171D0

Η λογική επιλογή θα ήταν να εκτελεσθεί ξανά η εφαρμογή γραμμή προς γραμμή για να βρεθεί το σημείο που καθορίζεται η τιμή στην διεύθυνση 00581890 που φαίνεται να καθορίζει την τιμή του EAX ή ακόμα καλύτερα, χρησιμοποιώντας ένα σημείο διακοπής υλικού στην διεύθυνση αυτή, να παρατηρηθεί οποιαδήποτε αλλαγή όμως αυτό δεν είναι εφικτό γιατί σε κάθε εκτέλεση της εφαρμογής η διεύθυνση αυτή αλλάζει. Ακόμα ο OllyDbg δεν βρίσκει την διεύθυνση αν πραγματοποιηθεί αναζήτηση για αυτήν κατά την αρχή της εκτέλεσης της εφαρμογής. Δοκιμάζοντας όμως με διαφορετικές συμβολοσειρές βλέπουμε

ότι η τιμή αλλάζει ανάλογα με το μέγεθος της συμβολοσειράς που εισάγουμε (είναι σχεδόν σίγουρη υπόθεση πλέον πως υπολογίζεται το πλήθος των χαρακτήρων της συμβολοσειράς) αλλά έχει καθοριστεί πολύ νωρίτερα και σε αυτό το σημείο αποθηκεύεται το αποτέλεσμα μιας προηγούμενης διαδικασίας. Τέλος, με την εισαγωγή ως ονόματος χρήστη μιας συμβολοσειράς με περισσότερους από 10 χαρακτήρες βλέπουμε ότι η τιμή του EAX ορίζεται όπως ήταν αναμενόμενο αναλόγως, η συνθήκη της διεύθυνσης 0040143B πλέον αληθεύει και εκτελείται ένα καινούριο τμήμα κώδικα.

Πρακτικά έχει αποδειχθεί πως ένα τμήμα της ταυτοποίησης αποτελείται από τον έλεγχο του μεγέθους της συμβολοσειράς που εισάγεται ως όνομα χρήστη. Στην C++ θα μπορούσε να είναι μια δομή του τύπου:

If (username.length() >= 10)

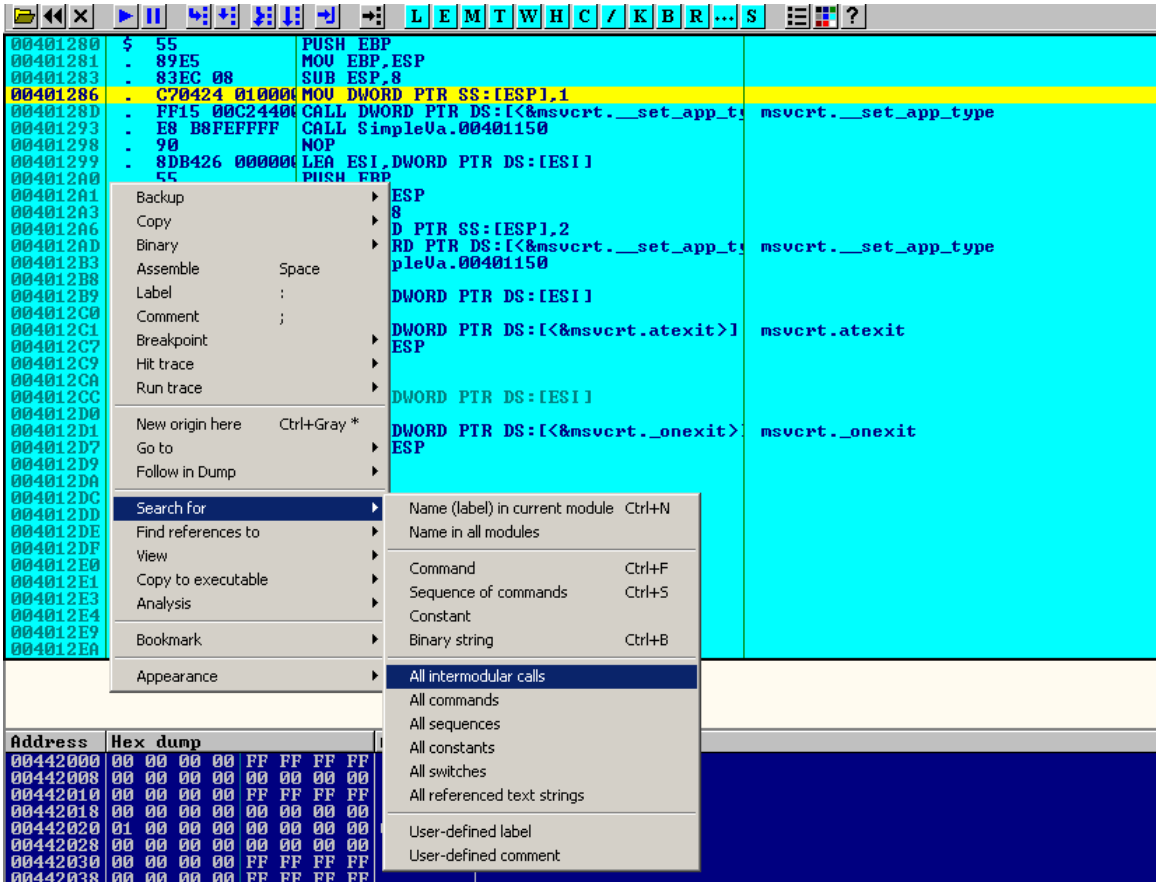
{εκτέλεσε κάποιες εντολές}

Else

{αποτυχία ταυτοποίησης }

Αν κάποιος θέλει όμως μπορεί να προχωρήσει και να ανακαλύψει το σημείο στο οποίο γίνεται ο πραγματικός έλεγχος για το μέγεθος της συμβολοσειράς. Αυτό είναι πολύ πιθανό να υπολογίζεται αμέσως μετά την εισαγωγή του ονόματος χρήστη. Για να επιτευχθεί αυτό είναι σημαντικό να σταματήσει η εκτέλεση της εφαρμογής αμέσως μετά το σημείο της εισαγωγής του ονόματος χρήστη και αν είναι δυνατόν αμέσως μετά το πάτημα του πλήκτρου «ENTER». Εκμεταλλευόμενος το γεγονός πως σχεδόν όλες οι εφαρμογές βασίζονται σε δυναμικές βιβλιοθήκες που παρέχονται από την Διεπαφή Προγραμματισμού Εφαρμογών των Windows (Windows API ή για συντομία WinAPI), ο μηχανικός λογισμικού ακόμα και όταν δεν γνωρίζει ποια ακριβώς κλήση είναι υπεύθυνη για την εισαγωγή των δεδομένων στην εφαρμογή, μπορεί να σταματήσει την ροή αρκετά κοντά στο ζητούμενο σημείο. Αυτό επιτυγχάνεται εισάγοντας ένα σημείο διακοπής σε κάθε κλήση που αφορά διαφορετική «μονάδα» (module) της εφαρμογής. Ο OllyDbg δίνει την δυνατότητα εύρεσης όλων αυτών των σημείων (κάνοντας δεξί κλικ στο παράθυρο του disassembler και επιλέγοντας διαδοχικά «search for» → «all intermodular calls» όπως φαίνεται στην Εικόνα 5.17) και εμφανίζει τα αποτελέσματα συγκεντρωτικά σε ξεχωριστό παράθυρο με πληροφορίες που αφορούν τη μονάδα στην οποία μεταφέρεται ο έλεγχος. Με αυτόν τον τρόπο διασφαλίζεται πως σε κάποιο σημείο της εκτέλεσης της εφαρμογής ο OllyDbg θα μεταφέρει τον έλεγχο στον χρήστη. Η πολύ καλή γνώση του WinAPI θα βοηθήσει τον

μηχανικό λογισμικού να κατανοήσει καλύτερα την εφαρμογή και να αποκτήσει τον έλεγχο της εφαρμογής στο σημείο που ακριβώς αυτός επιθυμεί.



Εικόνα 5.17: Αναζήτηση κλήσεων υπορουτίνων διαφορετικών μονάδων της εφαρμογής

Συνεχίζοντας λοιπόν την εκτέλεση της εφαρμογής ο έλεγχος μεταφέρεται στον χρήστη πάρα πολλές φορές μέχρι την εμφάνιση των μηνυμάτων που θα περίμενε από την εφαρμογή και την αναμονή για είσοδο του ονόματος χρήστη.

ΒΗΜΑ 2ο

Εισάγοντας ένα όνομα χρήστη μεγέθους δέκα ή περισσότερων χαρακτήρων βλέπουμε ότι αλλάζει η ροή εκτέλεσης όμως τελικά η ταυτοποίηση του χρήστη αποτυγχάνει, επομένως υπάρχει και άλλος κώδικας ταυτοποίησης του οποίου η λειτουργία πρέπει να αντιστραφεί και οι πληροφορίες δίδονται σταδιακά κατά την εκτέλεση σε μικρά τμήματα κώδικα αρχίζοντας στην διεύθυνση 00401478 όπως παρουσιάζεται στο Τμήμα Κώδικα 5.4 που εμφανίζεται για πρώτη φορά η συμβολοσειρά «admin».

```
00401433   .  E8 985D0100       CALL SimpleVa.004171D0
00401438   .  83F8 09            CMP EAX,9           ;  if string tested is over 10 chars jump
0040143B   .  77 0C              JA SHORT SimpleVa.00401449
0040143D   .  C745 90 00000>     MOV DWORD PTR SS:[EBP-70],0
00401444   .  E9 6F010000        JMP SimpleVa.004015B8
00401449   >  8D45 D8            LEA EAX,DWORD PTR SS:[EBP-28]
0040144C   .  C74424 0C 050>     MOV DWORD PTR SS:[ESP+C],5
00401454   .  C74424 08 020>     MOV DWORD PTR SS:[ESP+8],2
0040145C   .  8B55 08            MOV EDX,DWORD PTR SS:[EBP+8]
0040145F   .  895424 04          MOV DWORD PTR SS:[ESP+4],EDX
00401463   .  890424             MOV DWORD PTR SS:[ESP],EAX
00401466   .  C745 98 FFFFF>     MOV DWORD PTR SS:[EBP-68],-1
0040146D   .  E8 8E5D0100        CALL SimpleVa.00417200
00401472   .  83EC 04            SUB ESP,4
00401475   .  8D45 D8            LEA EAX,DWORD PTR SS:[EBP-28]
00401478   C74424 04 003>   MOV DWORD PTR SS:[ESP+4],SimpleVa.00443000          ;
ASCII "admin"
```

Τμήμα Κώδικα 5.4: Simple Validate (Συμβολοσειρά "admin")

Συνεχίζοντας την εκτέλεση της εφαρμογής πολλές φορές εμφανίζεται η συμβολοσειρά που έχει εισαχθεί ως όνομα χρήστη όμως ενδιαφέρον παρουσιάζει ο κώδικας από τη διεύθυνση 00417215 ως τη διεύθυνση 00417253 στο Τμήμα Κώδικα 5.5. Στο τμήμα αυτό, καταγράφοντας τις τιμές των καταχωρητών και κάποιων σημαντικών διευθύνσεων ενώ δίνοντας έμφαση στη διεύθυνση 00417247 που εμφανίζεται η συμβολοσειρά «ASCII "basic_string::substr"» μπορεί να συμπεράνει κανείς πως θα γίνει έλεγχος μόνο σε ένα τμήμα της συμβολοσειράς που εισήχθη ως όνομα χρήστη. Ακόμα ίσως είναι σημαντική η εμφάνιση των τιμών «2» και «5» στους καταχωρητές EAX και ESI αντίστοιχα.

```
00417215  |.  8B07            MOV EAX,DWORD PTR DS:[EDI]          ;  eax = username
00417217  |.  8B75 10         MOV ESI,DWORD PTR SS:[EBP+10]       ;  esi=2
0041721A  |.  3970 F4         CMP DWORD PTR DS:[EAX-C],ESI
```

```
0041721D  |.  72 28              JB SHORT SimpleVa.00417247
0041721F  |>  897424 08          MOV DWORD PTR SS:[ESP+8],ESI
00417223  |.  8B45 14            MOV EAX,DWORD PTR SS:[EBP+14]       ;  eax=5
00417226  |.  897C24 04          MOV DWORD PTR SS:[ESP+4],EDI
0041722A  |.  891C24             MOV DWORD PTR SS:[ESP],EBX
0041722D  |.  894424 0C          MOV DWORD PTR SS:[ESP+C],EAX        ;  0028FD8C = 5
00417231  |.  E8 5AAB0100        CALL SimpleVa.00431D90
00417236  |.  89D8               MOV EAX,EBX
00417238  |.  8B75 F8            MOV ESI,DWORD PTR SS:[EBP-8]
0041723B  |.  8B5D F4            MOV EBX,DWORD PTR SS:[EBP-C]
0041723E  |.  8B7D FC            MOV EDI,DWORD PTR SS:[EBP-4]
00417241  |.  89EC               MOV ESP,EBP
00417243  |.  5D                 POP EBP
00417244  |.  C2 0400            RETN 4
00417247  |>C70424 453344>  MOV DWORD PTR SS:[ESP],SimpleVa.00443345        ;  ASCII
"basic_string::substr"
0041724E  |.  E8 5521FFFF        CALL SimpleVa.004093A8
00417253  \.^ EB CA              JMP SHORT SimpleVa.0041721F
```

Τμήμα Κώδικα 5.5: Simple Validate (Substring)

Στη συνέχεια το όνομα χρήστη δέχεται περαιτέρω επεξεργασία και αφαιρούνται από αυτό τα δύο πρώτα γράμματα στη διεύθυνση 00431FB όπως φαίνεται στο Τμήμα Κώδικα 5.6.

```
00431DD7   .  8B12          MOV EDX,DWORD PTR DS:[EDX]              ;  edx = username
00431DD9   .  8945 8C       MOV DWORD PTR SS:[EBP-74],EAX
00431DDC   .  8B45 10       MOV EAX,DWORD PTR SS:[EBP+10]           ;  eax=2
00431DDF   .  3942 F4       CMP DWORD PTR DS:[EDX-C],EAX            ;  731890
[0000000A] = 2? check
00431DE2   .  8955 88       MOV DWORD PTR SS:[EBP-78],EDX           ;  0028FD00 =
username
00431DE5   .  72 7E         JB SHORT SimpleVa.00431E65
```

```
00431DE7  >  8B55 90        MOV EDX,DWORD PTR SS:[EBP-70]
00431DEA  .  8B75 10        MOV ESI,DWORD PTR SS:[EBP+10]
00431DED  .  8B5D 88        MOV EBX,DWORD PTR SS:[EBP-78]          ;ebx=username
00431DF0  .  8B0A           MOV ECX,DWORD PTR DS:[EDX]             ;ecx=username
00431DF2  .  8B7D 10        MOV EDI,DWORD PTR SS:[EBP+10]
00431DF5  .  8B45 14        MOV EAX,DWORD PTR SS:[EBP+14]          ;  eax = 5
00431DF8  .  8B51 F4        MOV EDX,DWORD PTR DS:[ECX-C]
00431DFB  .  01FB           ADD EBX,EDI                            ;  username is
trimmed by 2 first letters
00431DFD  .  29F2           SUB EDX,ESI
00431DFF  .  39C2           CMP EDX,EAX
00431E01  .  77 02          JA SHORT SimpleVa.00431E05
00431E03  .  89D0           MOV EAX,EDX
```

Τμήμα Κώδικα 5.6: Simple Validate (Αφαίρεση των δύο πρώτων γραμμάτων του ονόματος χρήστη)

Τέλος παρατηρεί κανείς τον σχηματισμό μιας νέας συμβολοσειράς αποτελούμενης από τους χαρακτήρες της συμβολοσειράς που εισήχθη ως όνομα χρήστη. Η νέα αυτή συμβολοσειρά όπως παρουσιάζεται στο Τμήμα Κώδικα 5.7 αποτελεί το τμήμα της αρχικής συμβολοσειράς από τον τρίτο χαρακτήρα ως και τον έβδομο.

```
0042F6E9  |.  89D8           MOV EAX,EBX
0042F6EB  |.  C6041E 00      MOV BYTE PTR DS:[ESI+EBX],0
0042F6EF  |> \8B5D F4        MOV EBX,DWORD PTR SS:[EBP-C]      ;  username substring[3-
7] is seen here
0042F6F2  |.  8B75 F8        MOV ESI,DWORD PTR SS:[EBP-8]      ;  esi = 0028FCE0 = 2
0042F6F5  |.  8B7D FC        MOV EDI,DWORD PTR SS:[EBP-4]      ;  edi = 0028FCE4 = 2
0042F6F8  |.  89EC           MOV ESP,EBP
0042F6FA  |.  5D             POP EBP
0042F6FB  |.  C3             RETN
```

Τμήμα Κώδικα 5.7: Simple Validate («Username.substring(2,5)»)

Ο κώδικας που παρουσιάστηκε παραπάνω θα μπορούσε να συνοψιστεί στην C++ σε μια δομή του τύπου:

```
#include <String.h>
‘
‘
‘
String username = «κώδικας εισαγωγής συμβολοσειρών»
String temp = username.substr(2, 5)
```

Στη συνέχεια όπως παρουσιάζεται στο Τμήμα Κώδικα 5.8 προετοιμάζεται το καθοριστικό κομμάτι για την ταυτοποίηση του χρήστη. Σε αυτό το σημείο διαδοχικά φορτώνονται οι τιμές της συμβολοσειράς του ονόματος χρήστη όπως αυτή έχει διαμορφωθεί μετά την επεξεργασία που υπέστη στα προηγούμενα στάδια (δηλαδή το τμήμα του αρχικού ονόματος χρήστη από τον τρίτο χαρακτήρα ως και τον έβδομο). Ακόμα φορτώνεται το πλήθος των χαρακτήρων που αποτελούν τη διαμορφωμένη συμβολοσειρά, δηλαδή ο αριθμός πέντε. Επίσης φορτώνεται η συμβολοσειρά «admin» και γίνεται κλήση στην υπορουτίνα «strlen» που υπολογίζει το πλήθος των χαρακτήρων μιας συμβολοσειράς.

```
00417280  /$  55              PUSH EBP
00417281  |.  89E5            MOV EBP,ESP
00417283  |.  83EC 28          SUB ESP,28
00417286  |.  895D F4          MOV DWORD PTR SS:[EBP-C],EBX     ; |
00417289  |.  8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]     ; |
0041728C  |.  8975 F8          MOV DWORD PTR SS:[EBP-8],ESI     ; |
0041728F  |.  897D FC          MOV DWORD PTR SS:[EBP-4],EDI     ; |
00417292  |.  8B00             MOV EAX,DWORD PTR DS:[EAX]       ; |eax = username[3-7]
00417294  |.  8945 E0          MOV DWORD PTR SS:[EBP-20],EAX    ;|  007318FC  =  username
[3,7]
00417297  |.  8B40 F4          MOV EAX,DWORD PTR DS:[EAX-C]     ; |eax = 5
```

```
0041729A  |.  8945 E4          MOV DWORD PTR SS:[EBP-1C],EAX    ; |
0041729D  |.  8945 F0          MOV DWORD PTR SS:[EBP-10],EAX    ; |
004172A0  |.  8B45 0C          MOV EAX,DWORD PTR SS:[EBP+C]     ; |eax = "admin"
004172A3  |.  890424           MOV DWORD PTR SS:[ESP],EAX       ; |28FD80 = "admin"
004172A6  |.  E8 35E7FFFF      CALL <JMP.&msvcrt.strlen>        ; \strlen
```

Τμήμα Κώδικα 5.8: Simple Validate (Φόρτωση συμβολοσειρών)

Παρακάτω, στο Τμήμα Κώδικα 5.9:Simple Validate (Σύγκριση Username.substring(2,5) με τη συμβολοσειρά «admin») παρουσιάζεται η φόρτωση του μετρητή ECX με την τιμή 5 που αντιστοιχεί στο μέγεθος της συμβολοσειράς που θα ελεγχθεί, του καταχωρητή ESI με το διαμορφωμένο όνομα χρήστη και του καταχωρητή EDI με τη συμβολοσειρά «admin» ενώ στη διεύθυνση **004172C6** υλοποιείται η σύγκριση μεταξύ των δύο συμβολοσειρών με την εντολή «**REPE CMPS**». Η εντολή «REPE» από την αγγλική λέξη «repeat» (επανάληψη) χρησιμοποιείται μαζί με την εντολή σύγκρισης συμβολοσειρών «CMPS» ώστε να επαναληφθεί η σύγκριση σε όλους τους χαρακτήρες που αποτελούν τη συμβολοσειρά έναν προς έναν. Όταν χρησιμοποιείται σε συνδυασμό με την εντολή «REPE» ο καταχωρητής ECX αποθηκεύει το πλήθος των χαρακτήρων της συμβολοσειράς.

```
004172AB  |.  8945 EC          MOV DWORD PTR SS:[EBP-14],EAX
004172AE  |.  89C3             MOV EBX,EAX
004172B0  |.  8D45 EC          LEA EAX,DWORD PTR SS:[EBP-14]
004172B3  |.  3B5D E4          CMP EBX,DWORD PTR SS:[EBP-1C]
004172B6  |.  72 03            JB SHORT SimpleVa.004172BB
004172B8  |.  8D45 F0          LEA EAX,DWORD PTR SS:[EBP-10]
004172BB  |>  FC               CLD
004172BC  |.  8B08             MOV ECX,DWORD PTR DS:[EAX]       ;  ecx = 5
004172BE  |.  8B75 E0          MOV ESI,DWORD PTR SS:[EBP-20]    ;  esi = username[3,7]
004172C1  |.  8B7D 0C          MOV EDI,DWORD PTR SS:[EBP+C]     ;  edi = "admin"
004172C4  |.  39C9             CMP ECX,ECX
004172C6  |.  F3:A6            REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

Τμήμα Κώδικα 5.9:Simple Validate (Σύγκριση Username.substring(2,5) με τη συμβολοσειρά «admin»)

Έτσι λοιπόν το Τμήμα Κώδικα 5.9 μπορεί να συνοψιστεί στην C++ στις εντολές:

```
#include <String.h>
‘
‘
‘
String username = «κώδικας εισαγωγής συμβολοσειρών»
String temp = username.substr(2, 5)
if (temp.compare("admin")=0)
        {αποτυχία ταυτοποίησης}
Else
        {εκτέλεσε κάποιες εντολές}
```

ΒΗΜΑ 3ο

Εισάγοντας ένα όνομα χρήστη μεγέθους δέκα ή περισσότερων χαρακτήρων με τη συμβολοσειρά «admin» να περιέχεται από τον τρίτο χαρακτήρα ως και τον έβδομο βλέπουμε ότι αλλάζει η ροή εκτέλεσης όμως τελικά η ταυτοποίηση του χρήστη και πάλι δεν επιτυγχάνεται. Επομένως είναι πολύ πιθανό να υπάρχουν περισσότεροι έλεγχοι.

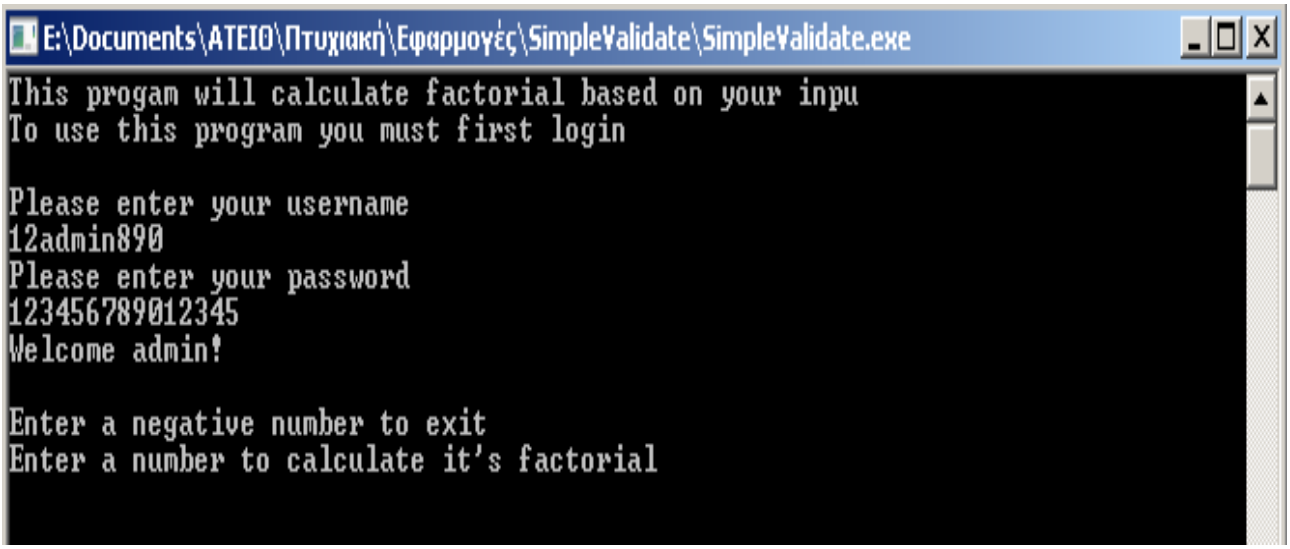
Συνεχίζοντας την εκτέλεση του κώδικα της εφαρμογής καταλήγει κανείς σε ένα πολύ ενδιαφέρον σημείο που παρουσιάζεται στο Τμήμα Κώδικα 5.10 και το οποίο με την πρώτη ματιά φαίνεται πολύ οικείο καθώς στο ΒΗΜΑ 1ο παρουσιάστηκε κάτι παρόμοιο στο Τμήμα Κώδικα 5.3: Simple Validate και όπως διαπιστώθηκε αφορούσε την σύγκριση του πλήθους των χαρακτήρων του ονόματος χρήστη. Επομένως φαντάζει πολύ πιθανό να συμβαίνει κάτι αντίστοιχο και με το συνθηματικό.

```
0040159E   . E8 2D5C0100    CALL SimpleVa.004171D0   ;  reminds us of previous statement about username length
004015A3   . 83F8 0E        CMP EAX,0E               ;  possibly checking password length
```

```
004015A6   . 77 09        JA SHORT SimpleVa.004015B1
```

Τμήμα Κώδικα 5.10: Simple Validate (Πιθανός έλεγχος του πλήθους χαρακτήρων του συνθηματικού)

Πράγματι εξετάζοντας την υπορουτίνα **«004171D0»** όπως και πριν διαπιστώνεται πως εμφανίζεται η συμβολοσειρά που εισήχθη ως συνθηματικό και το πλήθος των χαρακτήρων που την αποτελούν. Δοκιμάζοντας διαφορετικές συμβολοσειρές πιστοποιείται πως πραγματικά η τιμή του καταχωρητή EAX μεταβάλλεται ανάλογα με το πλήθος των χαρακτήρων του συνθηματικού. Για να επιτευχθεί τελικώς το «άλμα» της διεύθυνσης 004015A6 και να εκτελεστεί κανονικά η εφαρμογή όπως παρουσιάζεται στην *Εικόνα 5.18* πρέπει το πλήθος των χαρακτήρων του συνθηματικού να είναι μεγαλύτερο του 14_{10} ($0E_{16}$) και το όνομα χρήστη να αποτελείται από δέκα ή περισσότερους χαρακτήρες με την συμβολοσειρά «admin» να περιέχεται ξεκινώντας από τον τρίτο χαρακτήρα ως και τον έβδομο..



Εικόνα 5.18: Επιτυχής εκτέλεση με κλειδιά από την αντιστροφή του αλγόριθμου ταυτοποίησης

5.4.2 Γεννήτριες κλειδιών

Οι γεννήτριες κλειδιών είναι μικρές εφαρμογές οι οποίες με μια απλά σχεδιασμένη διεπαφή δημιουργούν και προσφέρουν στον χρήστη κλειδιά – στη συγκεκριμένη περίπτωση συνδυασμούς ονόματος χρήστη και συνθηματικού – τα οποία θεωρούνται από την εφαρμογή ως αυθεντικά. Πρόκειται ουσιαστικά για την υλοποίηση ενός αλγορίθμου που χρησιμοποιεί τη γνώση που αποκτήθηκε κατά την αντιστροφή του αλγόριθμου ταυτοποίησης.

Πολλές φορές δημιουργούνται εφαρμογές προς αντιστροφή αλλά με μόνη αποδεκτή λύση την επιτυχή δημιουργία μιας γεννήτριας κλειδιών ως εκπαιδευτικό υλικό στην ΑΜΛ. Αυτές οι εφαρμογές αποκαλούνται «KeygenMe».

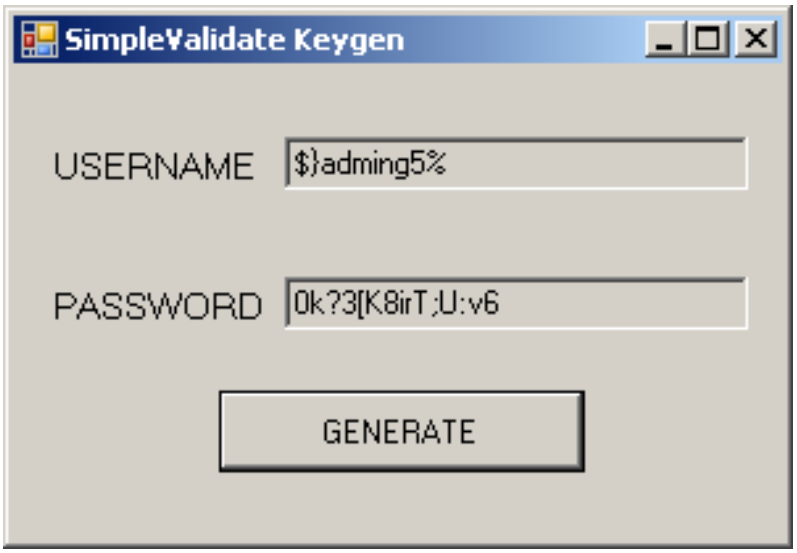
Έτσι λοιπόν στην Εφαρμογή SimpleValidate.cpp λύση θα αποτελούσε το Τμήμα Κώδικα 5.11: SimpleValidate (Γεννήτρια Κλειδιών) γραμμένο σε C#, μέρος από την Εφαρμογή SimpleValidateKeygen (βλ. Παράρτημα Α) το οποίο χειρίζεται την δημιουργία των τυχαίων κλειδιών.

```
char[] temp = new char[15];

Random rand = new Random();
for (int i = 0; i < 2; i++)
{
    int randomNumber = rand.Next(33, 126);
    temp[i] = (char) randomNumber;
}
temp[2]='a';
temp[3]='d';
temp[4]='m';
temp[5]='i';
temp[6]='n';
for (int i = 7; i < 10; i++)
{
    int randomNumber = rand.Next(33, 126);
    temp[i] = (char) randomNumber;
}
```

```
string s = new string(temp);
textBox1.Text = s;
Array.Clear(temp, 0, 15);

for (int i = 0; i < 15; i++)
{
    int randomNumber = rand.Next(33, 126);
    temp[i] = (char)randomNumber;
}
s = new string(temp);
textBox2.Text = s;
```

Τμήμα Κώδικα 5.11: SimpleValidate (Γεννήτρια Κλειδιών)

Τέλος μπορεί να δει κανείς στην Εικόνα 5.19 πως με το πάτημα ενός κουμπιού ο χρήστης έχει πλέον τη δυνατότητα να δημιουργεί άπειρους, τυχαίους συνδυασμούς ονομάτων χρήστη και συνθηματικών που θεωρούνται έγκυρα από την εφαρμογή.



Εικόνα 5.19: SimpleValidate (Γεννήτρια Κλειδιών)

5.5 Σύνοψη

Στην ενότητα αυτή συζητήθηκαν οι βασικές μεθοδολογίες που χρησιμοποιούνται για την επίτευξη μικρών αλλαγών σε εφαρμογές. Συγκεκριμένα παρουσιάστηκε μέσα από απλά παραδείγματα η ανάλυση της συμπεριφοράς μιας εφαρμογής και οι χαοτικές αλλαγές ροών εκτέλεσης που συμβαίνουν κατά την εκτέλεση αυτής, ενώ παράλληλα η ανίχνευση έδινε σε κάθε στάδιο πληροφορίες που αφορούσαν τιμές καταχωρητών, την εύρεση δομικών υπορουτίνων στον κώδικα της εφαρμογής καθώς και αναφορές σε άλλες υπορουτίνες.

Στη συνέχεια χρησιμοποιώντας τεχνικές όπως την εύρεση συμβολοσειρών και την εισαγωγή διακοπών σε κλήσεις που αφορούν διαφορετικές «μονάδες» της εφαρμογής έγινε εφικτή η στόχευση συγκεκριμένων σημείων στον κώδικα που αφορούσαν την εκτέλεση της εφαρμογής σε κάποια δεδομένη χρονική στιγμή. Με την καλή γνώση της συμβολικής γλώσσας έγιναν καλύτερα κατανοητές οι διεργασίες που έλαβαν χώρα κατά την εκτέλεση της εφαρμογής και έτσι δόθηκε η ευκαιρία στον μηχανικό λογισμικού να διορθώσει την συμπεριφορά της εφαρμογής με απλές ή προχωρημένου επιπέδου επιδιορθώσεις στον κώδικα. Τέλος έγινε ολοφάνερο πως μικρές αλλαγές σε μια γλώσσα υψηλού επιπέδου έχουν τεράστιες επιπτώσεις στην πολυπλοκότητα της εφαρμογής σε χαμηλό επίπεδο αλλά ακόμα και αυτό δεν ήταν αρκετό για να σταματήσει τον μηχανικό λογισμικού να πετύχει τον στόχο του.

Έτσι λοιπόν καταλαβαίνει κανείς πως ο σύνθετος κώδικας δεν είναι αρκετός να ασφαλίσει μια εφαρμογή από εξωτερικές παρεμβάσεις αν και δυσχεραίνει τη διαδικασία. Για αυτό πολλές εταιρείες που ασχολούνται με την προστασία του λογισμικού έχουν καταφύγει σε άλλες μεθόδους που στοχεύουν κυρίως τα εργαλεία που χρησιμοποιούνται στην ΑΜΛ.

Στο επόμενο κεφάλαιο θα παρουσιαστούν μέθοδοι όπως η αποτροπή αποσφαλμάτωσης (anti-debugging) καθώς και θα εξηγηθεί η λειτουργία των αποκαλούμενων ως «packer» ή «protector» εφαρμογών που στοχεύουν με διαφορετική φιλοσοφία στην προστασία των εφαρμογών.

6. Προστασία από την αντίστροφη μηχανική λογισμικού.

Εισαγωγή

Το λογικό επακόλουθο της ανάπτυξης των τεχνικών της ΑΜΛ και της συνεχόμενης αύξησης του αριθμού των μηχανικών λογισμικού που ασχολούνται με τον τομέα ήταν η προσπάθεια για εύρεση τρόπων προστασίας των εφαρμογών. Έτσι αναπτύχθηκαν πολλοί τρόποι για να εμποδίσουν τον μηχανικό λογισμικού να αντιστρέψει την λειτουργία των εφαρμογών, μερικοί πολύ απλοϊκοί και μερικοί που πραγματικά ανέβασαν το επίπεδο δυσκολίας.

Με μια πρώτη προσέγγιση στο θέμα κατευθείαν σκέφτεται κανείς πως οι βασικές εφαρμογές που θα έπρεπε να προστατέψουν τον κώδικα τους είναι κυρίως κακόβουλο λογισμικό ώστε να αποτραπεί η εύρεση τρόπων ανάλυσης τους και προστασίας από αυτες όμως στην πραγματικότητα πολλές εφαρμογές του εμπορίου έχουν σχεδιαστεί χρησιμοποιώντας παρόμοιες τεχνικές προσπαθώντας να προστατέψουν οι εταιρείες τα πνευματικά τους δικαιώματα. Τις τελευταίες τεχνικές προστασίας από την ΑΜΛ μπορεί να τις συναντήσει κανείς σε κακόβουλο λογισμικό και στην βιομηχανία βιντεοπαιχνιδιών.

Παρόλα αυτά όμως όσες μεθόδους και να χρησιμοποιήσει ο προγραμματιστής δεν μπορεί στην πραγματικότητα να καταστήσει αδύνατη την αντιστροφή μιας εφαρμογής καθώς κάθε είδους κώδικας που χρησιμοποιείται καταλήγει σε εκτελέσιμο κώδικα ο οποίος μπορεί να αντιστραφεί. Άρα ο μηχανικός λογισμικού μπορεί να αντιστρέψει κάθε είδους εφαρμογή δεδομένου χρόνου, ικανότητας και γνώσεως.

6.1 Αντί-αποσφαλμάτωση (Anti-Debugging).

Ένας από τους βασικούς τρόπους με τον οποίο οι προγραμματιστές δυσχεραίνουν το έργο του μηχανικού λογισμικού στην ΑΜΛ είναι η εισαγωγή κώδικα εντοπισμού εργαλείων αποσφαλμάτωσης. Στην Εφαρμογή IsDebuggerPresent.cpp έχει εισαχθεί κώδικας που ελέγχει αν η εκτέλεση της εφαρμογής συμβαίνει σε περιβάλλον κάποιου εργαλείου αποσφαλμάτωσης και παύει την εκτέλεση της ενώ σε αντίθετη περίπτωση η εφαρμογή εκτελείται κανονικά.

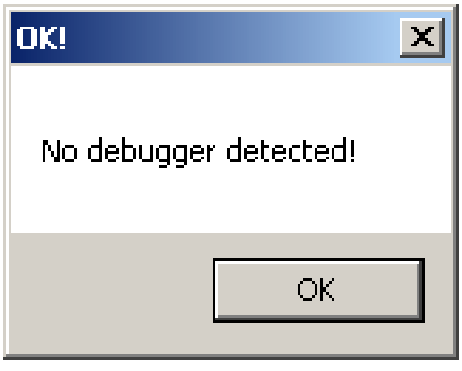
6.1.1 Ανάλυση της εφαρμογής

ΒΗΜΑ 1ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 2ο (βλ. ενότητα 5.2.1)

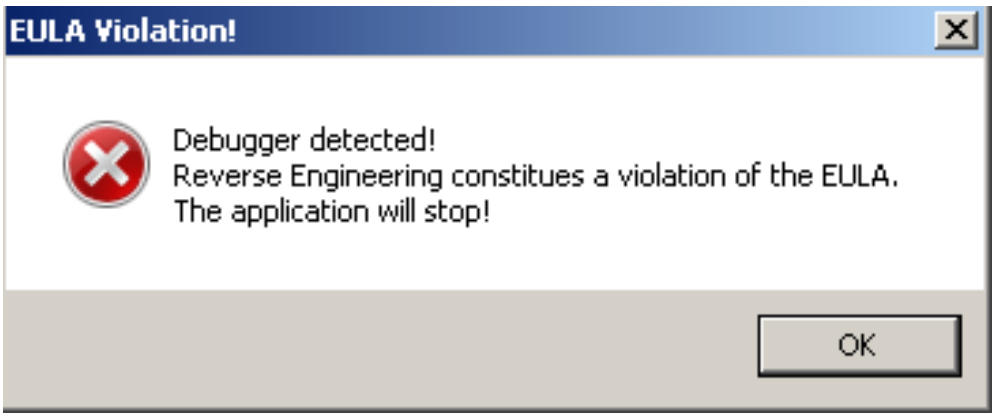
ΒΗΜΑ 3ο

Εκτελώντας την εφαρμογή παρατηρείται άμεσα η πρώτη διαφορά με ένα προειδοποιητικό μήνυμα το οποίο αναφέρει την μη ύπαρξη εργαλείου αποσφαλμάτωσης όπως φαίνεται στην Εικόνα 6.1 ενώ στη συνέχεια η εφαρμογή εκτελείται κανονικά.



Εικόνα 6.1: Εκτέλεση IsDebuggerPresent.cpp εκτός περιβάλλοντος εργαλείου αποσφαλμάτωσης

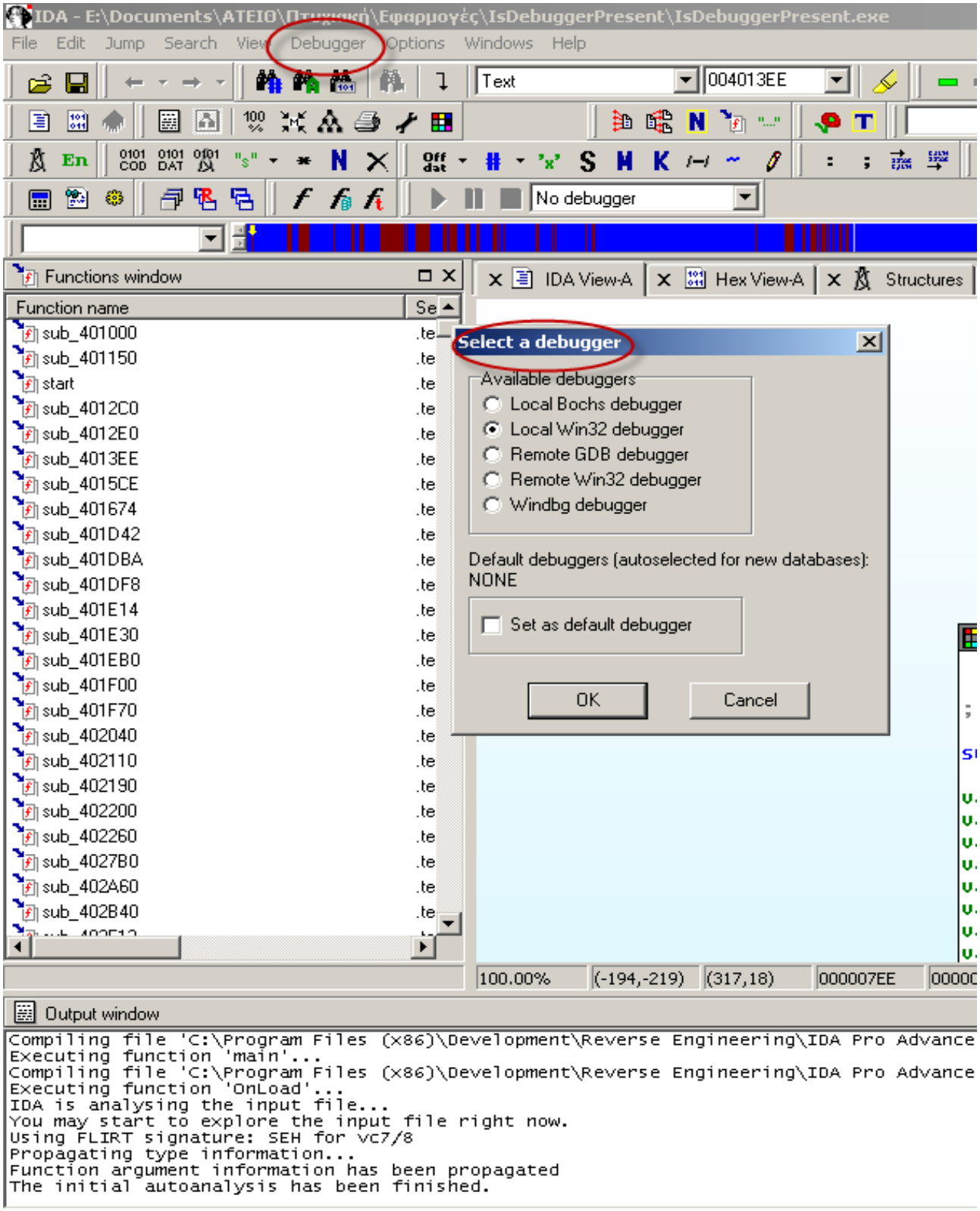
Αντίθετα όταν εκτελείται σε περιβάλλον εργαλείου αποσφαλμάτωσης εμφανίζει προειδοποιητικό μήνυμα και τερματίζει την εκτέλεση της όπως φαίνεται στην Εικόνα 6.2.



Εικόνα 6.2: Εκτέλεση εφαρμογής IsDebuggerPresent.cpp σε περιβάλλον εργαλείου αποσφαλμάτωσης

Επειδή κάποιες φορές οι προγραμματιστές εφαρμογών στοχεύουν συγκεκριμένα ένα εργαλείο αποσφαλμάτωσης, για παράδειγμα τον Olly Debugger θα γίνει μια παρατήρηση της συμπεριφοράς της εφαρμογής και σε κάποιο άλλο εργαλείο (IDA Pro σε αυτήν την περίπτωση).

Ανοίγοντας την εφαρμογή με το εργαλείο αποσφαλμάτωσης IDA Pro και επιλέγοντας τον Local Win32 Debugger από το παράθυρο διαλόγου επιλογής Debugger (Debugger → Select Debugger) όπως φαίνεται στην Εικόνα 6.3 και στην συνέχεια εκτελώντας την εφαρμογή σε αυτό το περιβάλλον παρατηρεί κανείς πως το αποτέλεσμα είναι το ίδιο και εμφανίζεται πάλι το ενοχλητικό μήνυμα που παρουσιάστηκε στην Εικόνα 6.2 επομένως επιστρέφουμε στον Olly Debugger για βαθύτερη ανάλυση της συμπεριφοράς της εφαρμογής σε επίπεδο συμβολικής γλώσσας.



Εικόνα 6.3: Επιλογή Debugger στο εργαλείο αποσφαλμάτωσης IDA Pro

ΒΗΜΑ 4ο

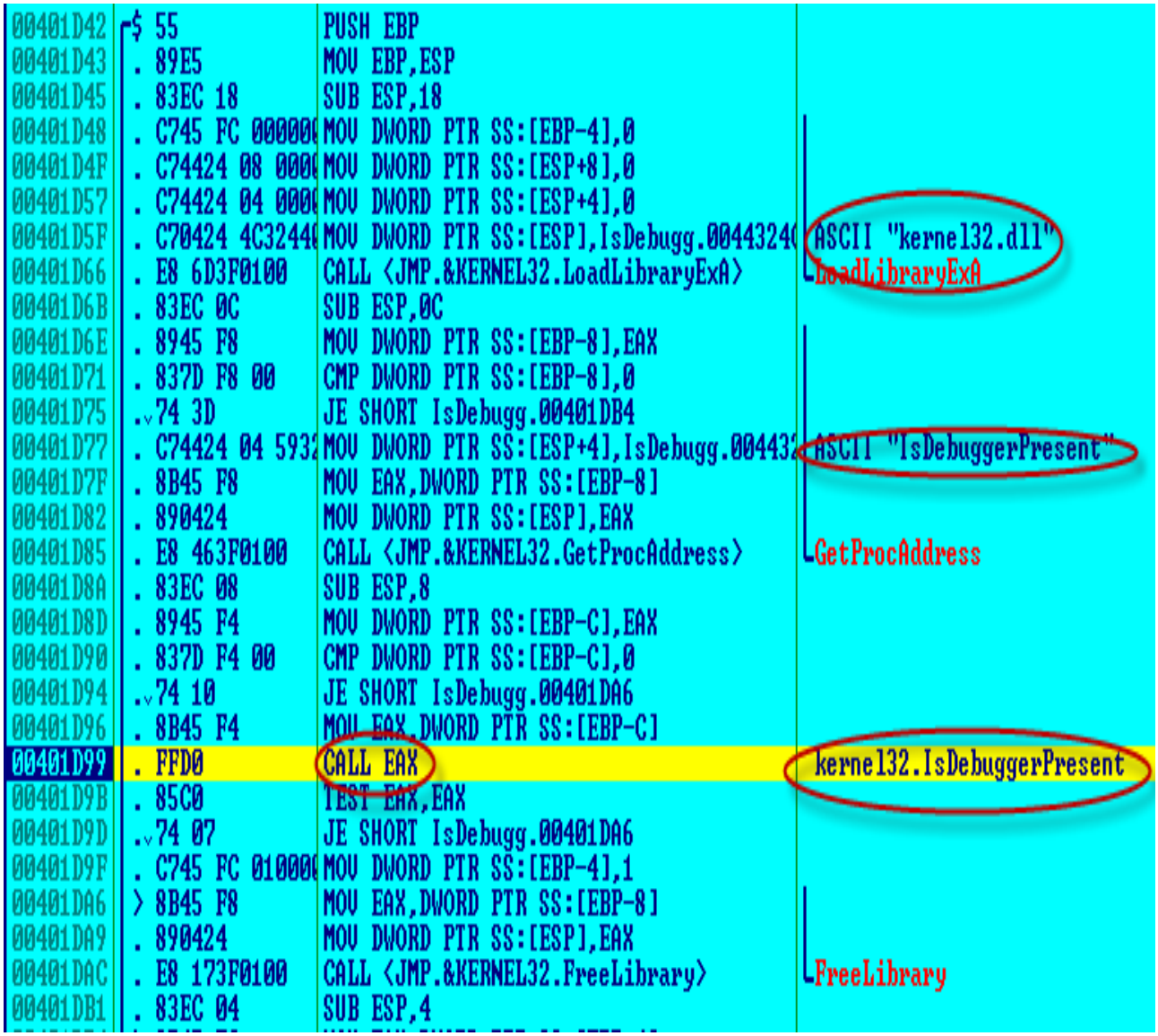
Χρησιμοποιώντας την τεχνική της εύρεσης συμβολοσειρών και αναζητώντας το κείμενο του μηνύματος όπως παρουσιάστηκε στην Εικόνα 6.2 οδηγείται ο μηχανικός λογισμικού στο Τμήμα Κώδικα 6.1: Κλήση IsDebuggerPresent και εμφάνιση μηνύματος αποτυχίας στο οποίο εμφανίζονται τα μηνύματα αποτυχίας και επιτυχίας που αφορούν την ύπαρξη εργαλείου αποσφαλμάτωσης.

```
004016B9  . E8 84060000   CALL IsDebugg.00401D42
004016BE  . 85C0          TEST EAX,EAX
004016C0  .v74 37         JE SHORT IsDebugg.004016F9
004016C2  . E8 E1450100   CALL <JMP.&USER32.GetForegroundWindow>   [GetForegroundWindow
004016C7  . C74424 0C 100(MOV DWORD PTR SS:[ESP+C],10
004016CF  . C74424 08 5B30(MOV DWORD PTR SS:[ESP+8],IsDebugg.004430   ASCII "EULA Violation!"
004016D7  . C74424 04 6C30(MOV DWORD PTR SS:[ESP+4],IsDebugg.004430   ASCII "Debugger detected!@Reverse Engineering constitues a viola
004016DF  . 890424        MOV DWORD PTR SS:[ESP],EAX
004016E2  . E8 C9450100   CALL <JMP.&USER32.MessageBoxA>            MessageBoxA
004016E7  . 83EC 10       SUB ESP,10
004016EA  . C785 6CFFFFFF MOV DWORD PTR SS:[EBP-94],0
004016F4  .vE9 2A060000   JMP IsDebugg.00401D23
004016F9  > C785 74FFFFFF MOV DWORD PTR SS:[EBP-8C],-1
00401703  . E8 A0450100   CALL <JMP.&USER32.GetForegroundWindow>   [GetForegroundWindow
00401708  . C74424 0C 000(MOV DWORD PTR SS:[ESP+C],0
00401710  . C74424 08 D230(MOV DWORD PTR SS:[ESP+8],IsDebugg.004430   ASCII "OK!"
00401718  . C74424 04 D630(MOV DWORD PTR SS:[ESP+4],IsDebugg.004430   ASCII "No debugger detected!"
00401720  . 890424        MOV DWORD PTR SS:[ESP],EAX
00401723  . E8 88450100   CALL <JMP.&USER32.MessageBoxA>            MessageBoxA
00401728  . 83EC 10       SUB ESP,10
0040172B  . 8D45 D8       LEA EAX,DWORD PTR SS:[EBP-28]
```

Τμήμα Κώδικα 6.1: Κλήση IsDebuggerPresent και εμφάνιση μηνύματος αποτυχίας

Όπως και τις προηγούμενες φορές αμέσως πριν υπάρχει μια κλήση σε μια υπορουτίνα, ένας τυπικός έλεγχος και μια κοντινή αλλαγή ροής εκτέλεσης. Προφανώς η αλλαγή της εντολής «JE» στην διεύθυνση 004016C0 σε «JMP» είναι αρκετή για να προσπεράσει το μήνυμα αποτυχίας και να οδηγήσει στην σωστή ροή εκτέλεσης της εφαρμογής όμως όπως έχει αναφερθεί σε προηγούμενη ενότητα αυτό αποτελεί μια πολύ

απλοϊκή λύση ευάλωτη σε διπλο-ελέγχους και καλό θα ήταν να ερευνηθεί βαθύτερα ο λόγος που δεν εκτελείται η εφαρμογή σε περιβάλλον αποσφαλμάτωσης. Για να επιτευχθεί αυτό θα πρέπει να γίνει εκτέλεση (με Step-Into) της υπορουτίνας 00401D42. Ήδη με την είσοδο στην υπορουτίνα και όπως παρουσιάζονται στο Τμήμα Κώδικα 6.2: Η υπορουτίνα 00401D42 ήδη κάποια σημεία δείχνουν ύποπτα.

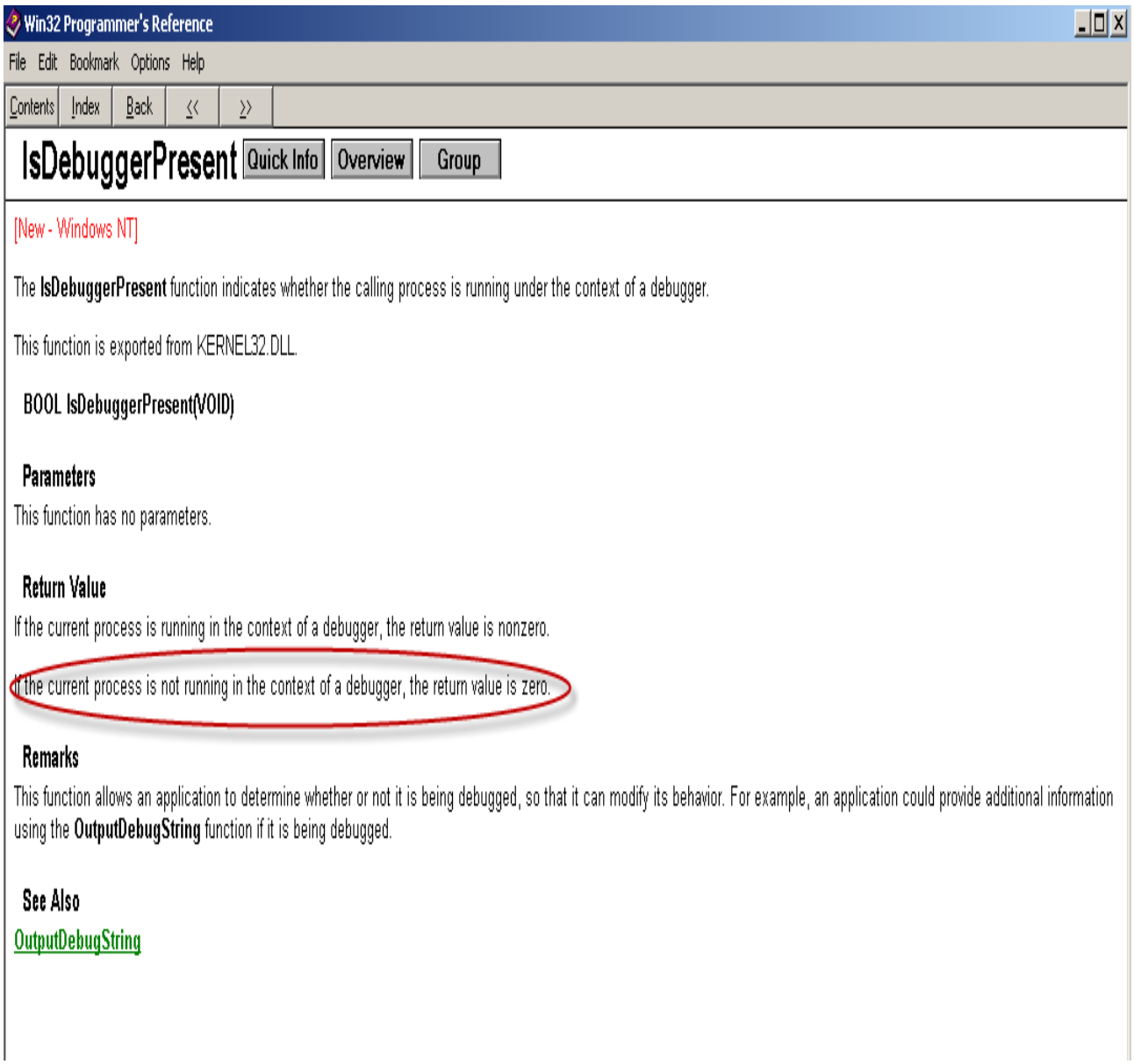


Τμήμα Κώδικα 6.2: Η υπορουτίνα 00401D42

Αρχικά στη διεύθυνση 00401D5F φορτώνεται η δυναμική βιβλιοθήκη «kernel32.dll». Στη διεύθυνση 00401D77 εμφανίζεται η συμβολοσειρά «kernel32.IsDebuggerPresent» που προφανώς αναφέρεται σε κάποια υπορουτίνα της διεπαφής προγραμματισμού

εφαρμογών των Windows (WinAPI) ενώ στη διεύθυνση 00401D99 γίνεται η κλήση της υπορουτίνας αυτής.

Σε αυτό το σημείο καλό θα ήταν να ανατρέξει κανείς στην τεκμηρίωση της Microsoft για το WinAPI (καθώς η υπορουτίνα αποτελεί κομμάτι της δυναμικής βιβλιοθήκης Kernel32.dll) και να εντοπίσει περισσότερες πληροφορίες για την υπορουτίνα «IsDebuggerPresent».



Εικόνα 6.4: Win32 API (IsDebuggerPresent)

Η Εικόνα 6.4 παρουσιάζει τις πληροφορίες που παρέχονται από την Microsoft για την υπορουτίνα «IsDebuggerPresent) και ειδικότερα επεξηγεί πως έχει σχέση με το αν εκτελείται η εφαρμογή σε περιβάλλον αποσφαλμάτωσης ή όχι. Ακόμα παρέχει την

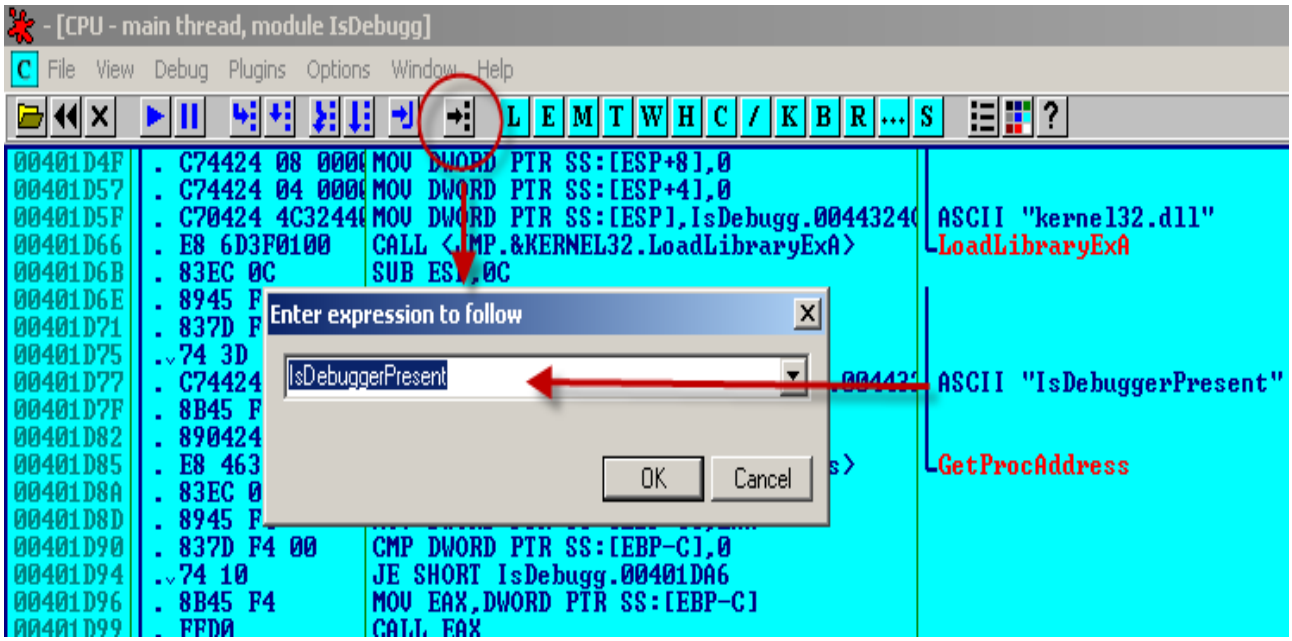
σημαντική πληροφορία πως αν η εφαρμογή δεν εκτελείται σε περιβάλλον αποσφαλμάτωσης η τιμή που επιστρέφεται είναι ίση με το μηδέν. Επομένως στον καταχωρητή EAX θα τοποθετηθεί η τιμή μηδέν μόνο εάν η εφαρμογή εκτελείται κανονικά.

6.1.2 Παράκαμψη της προστασίας.

Η βασικότερη τεχνική που χρησιμοποιείται σε πολλές περιπτώσεις προστασίας από την αποσφαλμάτωσης είναι αυτή που θα χρησιμοποιηθεί σε αυτήν την περίπτωση και πρόκειται για επιδιόρθωση του κώδικα μέσα στην υπορουτίνα «IsDebuggerPresent».

ΒΗΜΑ 1ο

Επιλέγοντας από την γραμμή εργαλείων το κουμπί «Go to address in disassembler» εμφανίζεται ένα παράθυρο διαλόγου που ζητάει να εισάγει ο μηχανικός λογισμικού μια έκφραση για να την «ακολουθήσει», δηλαδή να βρει μέσα στην εφαρμογή που ακριβώς βρίσκεται. Σε αυτό το παράθυρο διαλόγου πρέπει να εισαχθεί το όνομα της υπορουτίνας «IsDebuggerPresent» όπως στην



Εικόνα 6.5: Παράθυρο διαλόγου (Follow Expression)

Επιλέγοντας «OK» μεταφέρεται ο μηχανικός λογισμικού στην μονάδα η οποία περιέχει την υπορουτίνα «IsDebuggerPresent» και συνεχίζοντας την εκτέλεση μετά από δύο υποχρεωτικές αλλαγές ροής εκτέλεσης εμφανίζεται το Τμήμα Κώδικα 6.3 στο οποίο καθορίζεται η τιμή του καταχωρητή EAX (στη συγκεκριμένη περίπτωση EAX =1).

```
77082D6F > 64:A1 18000000    MOV EAX,DWORD PTR FS:[18]
77082D75   8B40 30           MOV EAX,DWORD PTR DS:[EAX+30]
77082D78   0FB640 02         MOVZX EAX,BYTE PTR DS:[EAX+2]      ; EAX = 1
```

Τμήμα Κώδικα 6.3: IsDebuggerPresent (Καθορισμός της τιμής του EAX)

Επομένως, για να αποφευχθεί ο έλεγχος και σύμφωνα με τις πληροφορίες που παρείχε η Microsoft αρκεί να αλλαχθεί το Τμήμα Κώδικα 6.3 όπως στο Τμήμα Κώδικα 6.4 ώστε ο EAX να παίρνει πάντα την τιμή 0 και να επιστρέφει η εφαρμογή στην προηγούμενη ροή εκτέλεσης.

```
77082D6F > B8 00000000       MOV EAX,0
77082D74   C3                RETN
77082D75   8B40 30           MOV EAX,DWORD PTR DS:[EAX+30]
77082D78   0FB640 02         MOVZX EAX,BYTE PTR DS:[EAX+2]      ; EAX = 1
```

Τμήμα Κώδικα 6.4: Τροποποίηση της υπορουτίνας IsDebuggerPresent

6.2 «Packers» και «Protectors»

Με τον όρο «packing» στην επιστήμη της πληροφορικής αναφερόμαστε σε οποιαδήποτε μέθοδο συμπίεσης αρχείων, όμως ειδικότερα στην ΑΜΛ ο όρος χρησιμοποιείται κυρίως για την διαδικασία συμπίεσης εκτελέσιμων αρχείων. Τα αρχεία αυτά συμπιέζονται ενώ παράλληλα περιέχουν και ένα τμήμα εκτελέσιμου κώδικα (stub) για την αποσυμπίεση τους ώστε να εκτελούνται χωρίς προβλήματα από το λειτουργικό σύστημα. Ο εκτελέσιμος κώδικας αυτός αποτελεί ένα εκτελέσιμο αρχείο το οποίο συνενώνεται με το εκτελέσιμο αρχείο της εφαρμογής ώστε να δημιουργηθεί ένα μοναδικό αρχείο το οποίο δεν μπορεί κανείς να διακρίνει άμεσα αν είναι συμπιεσμένο. Επομένως όταν ο κώδικας αυτός εκτελεστεί το εκτελέσιμο αρχείο της εφαρμογής θα αποσυμπιεστεί στην κανονική του μορφή και ο έλεγχος θα μεταφερθεί στο αρχικό σημείο εισόδου του εκτελέσιμου. Τα εργαλεία που επιτελούν αυτόν το σκοπό καλούνται «packers».

Τα πλεονεκτήματα της συμπίεσης αυτής είναι προφανή και συμπεριλαμβάνουν τα εξής:

1. Μείωση του χώρου που απαιτείται για την αποθήκευση του εκτελέσιμου αρχείου το οποίο έχει μεγάλη αξία όταν πρόκειται να διαδοθεί το αρχείο μέσω του διαδικτύου.
2. Μείωση του χρόνου που απαιτείται γα την μεταφορά του εκτελέσιμου κώδικα του αρχείου από τον σκληρό δίσκο στην μνήμη.
3. Δυσχεραίνει τη διαδικασία της επιδιόρθωσης του εκτελέσιμου αρχείου από άτομα που μόλις άρχισαν να ασχολούνται με την ΑΜΛ καθώς για να μπορέσει κάποιος να αλλάξει τον κώδικα του μέχρι πρότινος συμπιεσμένου εκτελέσιμου πρέπει πρώτα να το αποσυμπιέσει και να το επαναφέρει στην αρχική του μορφή.

Τα βασικά μειονεκτήματα τέτοιων εργαλείων είναι:

1. Ο χρόνος που απαιτείται για την προετοιμασία της εκτέλεσης της εφαρμογής αυξάνεται καθώς σε αυτόν προστίθεται ο χρόνος που απαιτείται για την αποσυμπίεση του εκτελέσιμου αρχείου.
2. Το κόστος της εφαρμογής αυξάνεται.
3. Αντι-ιικό λογισμικό μπορεί να αντιμετωπίσει το αρχείο ως κακόβουλο λογισμικό καθώς η τεχνική αυτή είναι πολύ συνηθισμένη πρακτική σε ιούς.

Η αδυναμία τους να προστατέψουν την εφαρμογή από την ΑΜΛ προκύπτει από το γεγονός πως από τη φύση τους, σε κάποιο σημείο της εκτέλεσης, πρέπει να αποσυμπιέσουν το εκτελέσιμο αρχείο για να μεταφέρουν τον έλεγχο στο αρχικό σημείο εισόδου του. Εκείνη ακριβώς τη στιγμή μπορεί κάποιος να αποθηκεύσει το αποσυμπιεσμένο πλέον εκτελέσιμο αρχείο στον δίσκο χωρίς τον κώδικα που το συμπίεσε αρχικά.

Αντίθετα με τον όρο «protectors» αναφερόμαστε σε εφαρμογές που λειτουργούν με παρόμοιο σκεπτικό αλλά στοχεύουν αποκλειστικά στην προστασία του εκτελέσιμου από τις τεχνικές της ΑΜΛ. Το πρόβλημα τους όμως είναι πως αυξάνουν κατά πολύ το μέγεθος της εφαρμογής (μέχρι και 600% του αρχικού μεγέθους) στην προσπάθεια να αποκλείσουν κάθε τρόπο πρόσβασης καθώς και τον χρόνο εκτέλεσης της εφαρμογής. Ακόμα ιστορικά δεν έχει βρεθεί κάποια εφαρμογή αυτού του τύπου που να μην έχει αντιστραφεί η

λειτουργία της αργά ή γρήγορα. Επίσης οι υψηλού επιπέδου προστασίες αποτελούν πρωταρχικό στόχο για τις ομάδες πειρατείας λογισμικού καθώς τις αντιμετωπίζουν ως προκλήσεις. Τέλος οι προγραμματιστές κακόβουλου λογισμικού προσπαθούν να το κρύβουν σε προστατευμένα αρχεία για να δυσκολεύουν την αναγνώριση τους από τους ευριστικούς αλγορίθμους αντι-ιικού λογισμικού. Σαν αποτέλεσμα αυξάνονται οι περιπτώσεις που αναγνωρίζονται ασφαλείς εμπορικές εφαρμογές ψευδώς ως κακόβουλο λογισμικό.

Έτσι μπορεί κανείς να πει πως είναι εξαιρετικά αμφιλεγόμενο το αν προσφέρουν αρκετή προστασία ανά μονάδα κόστους ή καλύτερα αν τα θετικά τους στοιχεία υπερσκελίζουν τα αρνητικά ενώ αντίθετα οι packers αναμφισβήτητα προσφέρουν πολλά θετικά στοιχεία στην εφαρμογή.

6.3 Αποσυμπίεση (unpacking) ενός εκτελέσιμου αρχείου.

Στην ΑΜΛ ο καλύτερος τρόπος για να παρουσιαστούν οι τεχνικές που χρησιμοποιούνται είναι όπως πάντα μέσα από παραδείγματα εφαρμογών. Η εφαρμογή στόχος θα είναι το εκτελέσιμο αρχείο «UnPackMe_CrypKeySDK5.7.exe» το οποίο όπως φανερώνει και το όνομα του έχει συμπιεστεί με την εφαρμογή «CrypKey 5.7» και έχει δημιουργηθεί αποκλειστικά για να αποσυμπιεστεί.

6.3.1 Ανάλυση της εφαρμογής.

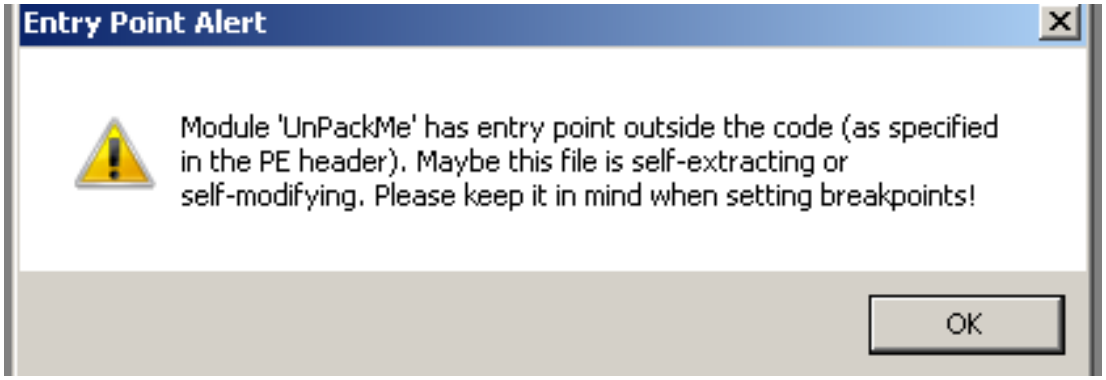
ΒΗΜΑ 1ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 2ο (βλ. ενότητα 5.2.1)

ΒΗΜΑ 3ο

Ανοίγοντας το εκτελέσιμο αρχείο από την φόρτωση του ήδη ο OllyDbg παρουσιάζει ήδη το πρώτο προειδοποιητικό μήνυμα όπως φαίνεται στην Εικόνα 6.6. Ενημερώνει πως

το σημείο εισόδου του αρχείου βρίσκεται εκτός του τμήματος κώδικα όπως αυτό δηλώνεται στην κεφαλή του φορητού εκτελέσιμου. Επίσης προειδοποιεί πως το αρχείο μπορεί να αποσυμπιέζεται μόνο του ή να τροποποιεί τον κώδικα του κατά την εκτέλεση. Σαν αποτέλεσμα ο μηχανικός λογισμικού πρέπει να είναι πολύ προσεκτικός όταν εισάγει σημεία διακοπής στον κώδικα.



Εικόνα 6.6: Προειδοποίηση OllyDbg για συμπιεσμένο εκτελέσιμο αρχείο

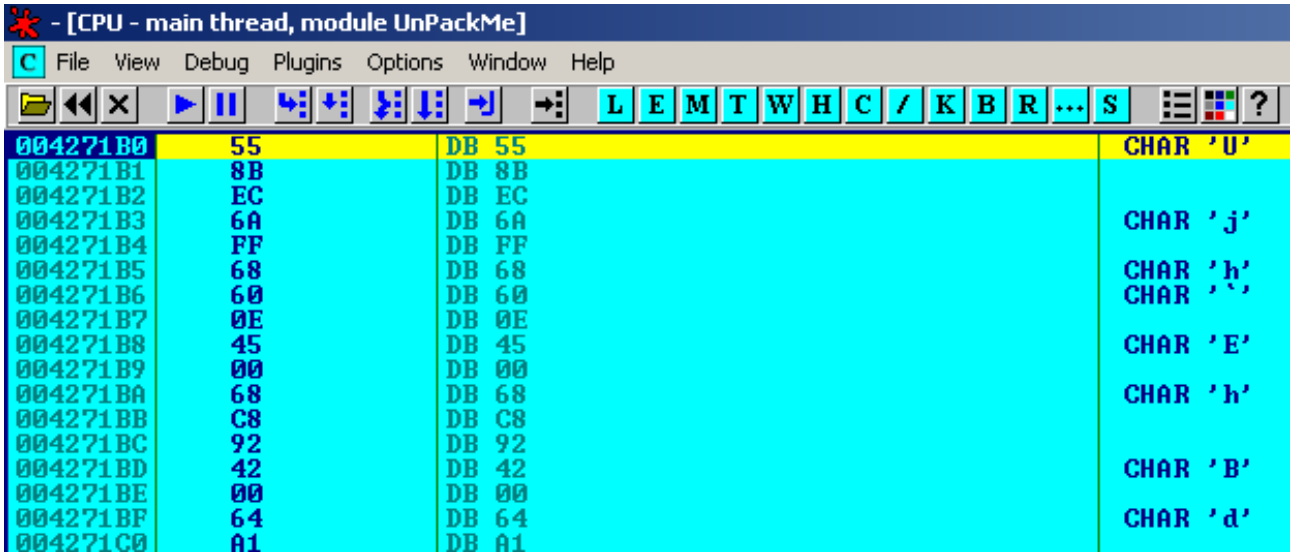
Επιλέγοντας «OK» ο OllyDbg μεταφέρει τον έλεγχο στον χρήστη σταματώντας την εκτέλεση στο σημείο εισόδου του κώδικα αποσυμπίεσης του εκτελέσιμου. Σε αυτό το σημείο καλό θα ήταν να μελετηθεί το Τμήμα Κώδικα 6.5 το οποίο αποτελεί και τον κώδικα αποσυμπίεσης του εκτελέσιμου αλλά στην συγκεκριμένη εφαρμογή ο κώδικας είναι απλός. Μετά από μερικές κλήσεις σε υπορουτίνες ακολουθεί μια αλλαγή ροής εκτέλεσης και μια εντολή επιστροφής.

```
0046B6DE >  E8 39000000      CALL UnPackMe.0046B71C
0046B6E3    E8 A40A0000      CALL UnPackMe.0046C18C
0046B6E8    6A 01            PUSH 1
0046B6EA    E8 09020000      CALL UnPackMe.0046B8F8
0046B6EF    A1 49B64600      MOV EAX,DWORD PTR DS:[46B649]
0046B6F4    83F8 01          CMP EAX,1
0046B6F7    74 06            JE SHORT UnPackMe.0046B6FF
0046B6F9  - FF25 14B04600    JMP DWORD PTR DS:[46B014]
0046B6FF    C3               RETN
```

Τμήμα Κώδικα 6.5: Κώδικας που αποσυμπιέζει την εφαρμογή

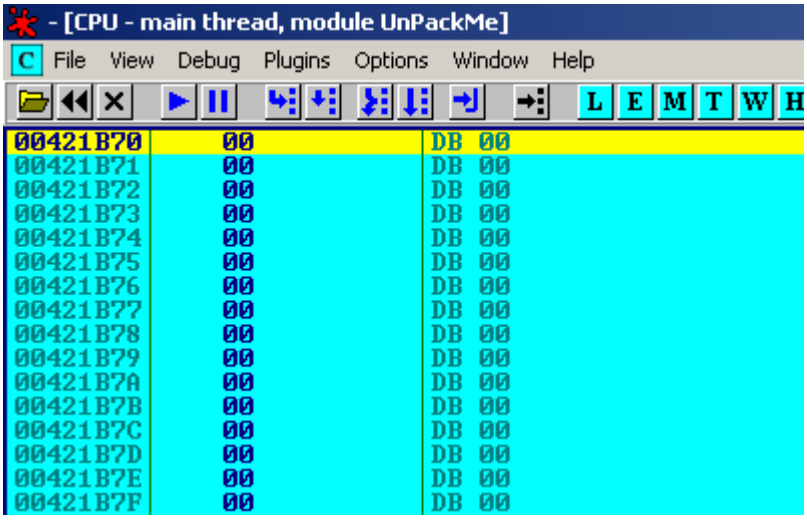
«UnPackMe_CrypKeySDK5.7.exe».

Εκτελώντας τον κώδικα γραμμή προς γραμμή (με Step Over) παρατηρεί κανείς πως μετά την αλλαγή ροής της διεύθυνσης 0046B6F9 μεταφέρεται ο χρήστης σε ένα τμήμα κώδικα το οποίο δεν έχει αναγνωριστεί ακόμα όπως φαίνεται στην Εικόνα 6.7: Μη αναγνωρισμένο τμήμα κώδικα ή καλύτερα αναγνωρισμένο ως τμήμα δεδομένων.



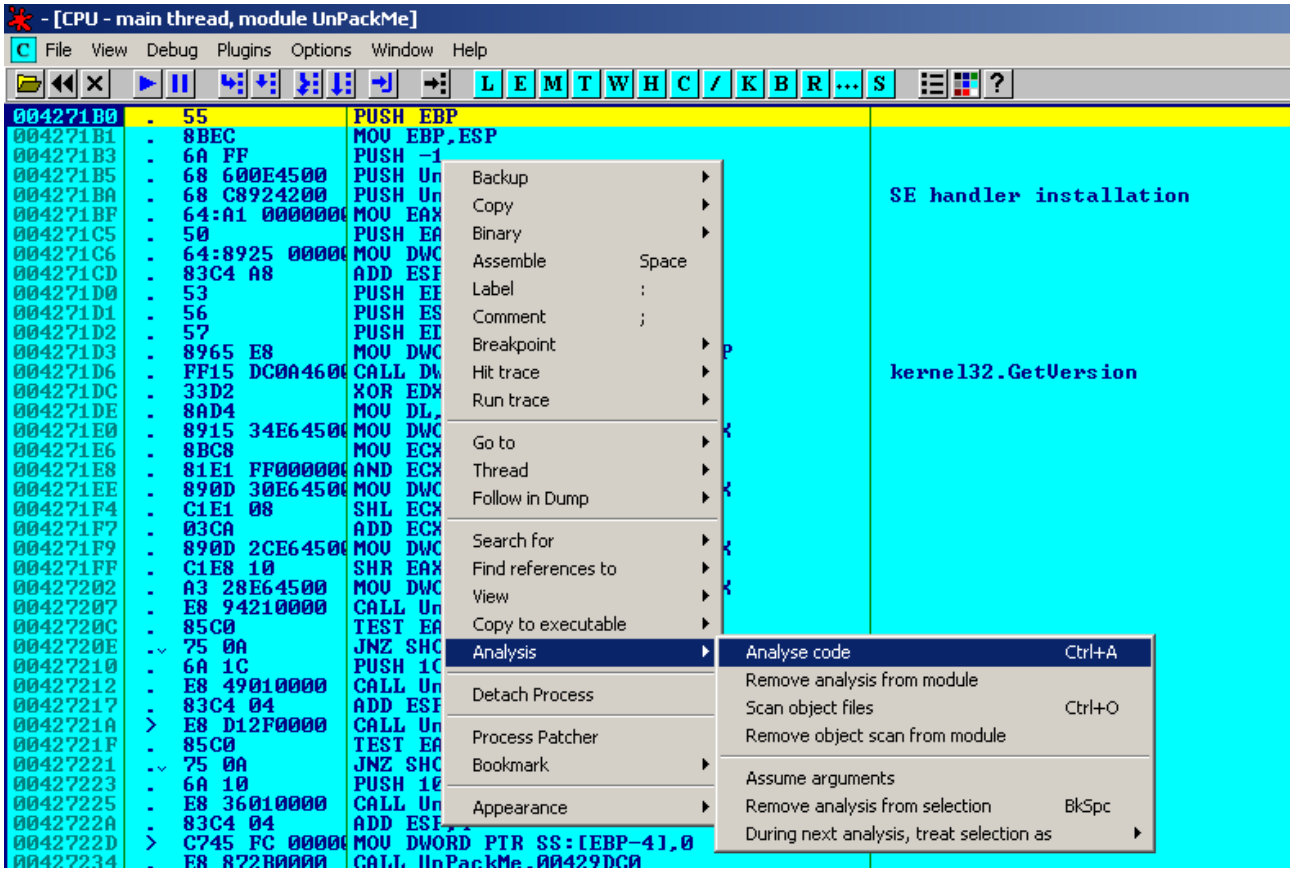
Εικόνα 6.7: Μη αναγνωρισμένο τμήμα κώδικα

Αυτό συμβαίνει γιατί κατά την φόρτωση του εκτελέσιμου και τη διακοπή στο σημείο εισόδου αν κάποιος ελέγξει τη διεύθυνση 004271B0 θα διαπιστώσει πως δεν υπήρχε κώδικας όπως φαίνεται στην Εικόνα 6.8: Κώδικας που λείπει από το εκτελέσιμο κατά την φόρτωση του.



Εικόνα 6.8: Κώδικας που λείπει από το εκτελέσιμο κατά την φόρτωση του

Επομένως ο μηχανικός λογισμικού πρέπει να ζητήσει από τον OllyDbg να αναλύσει για άλλη μια φορά τον κώδικα (επιλέγοντας με δεξί κλικ → Analysis → Analyse code) ώστε να μπορέσει να μελετήσει τον κώδικα που δημιουργήθηκε όπως παρουσιάζεται στην Εικόνα 6.9.



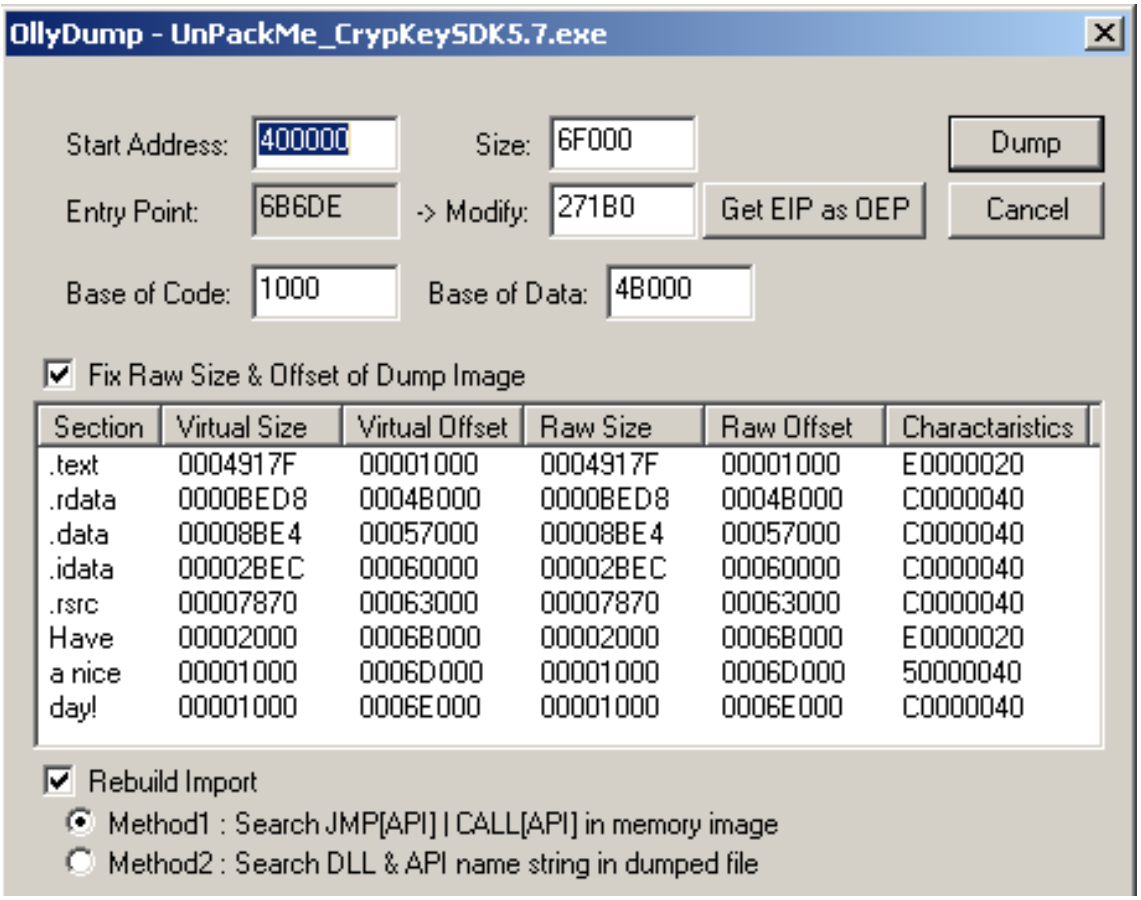
Εικόνα 6.9: Ο κώδικας που δημιουργήθηκε από την ρουτίνα αποσυμπίεσης του εκτελέσιμου

Αυτό ακριβώς το σημείο κώδικα είναι και το αυθεντικό σημείο εισόδου της εφαρμογής, δηλαδή η διεύθυνση 004271B0.

6.3.2 Αποθήκευση της αποσυμπιεσμένης εφαρμογής (dumping).

Εφόσον έχει εντοπιστεί το αυθεντικό σημείο εισόδου της εφαρμογής και έχει ολοκληρωθεί η εκτέλεση του κώδικα αποσυμπίεσης του εκτελέσιμου πρέπει να αποθηκευθεί το ολοκληρωμένο αρχείο. Επειδή όμως ο OllyDbg δεν έχει ενσωματωμένη αυτήν την λειτουργία απαιτείται μια πρόσθετη βιβλιοθήκη που αναλαμβάνει αυτή τη

λειτουργία. Η πρόσθετη βιβλιοθήκη OllyDump.dll επεκτείνει τις δυνατότητες του OllyDbg και έτσι σε αυτό το σημείο ο μηχανικός λογισμικού μπορεί πολύ απλά να αποθηκεύσει το νέο εκτελέσιμο (επιλέγοντας με δεξί κλικ → Dump Debugged Process). Στο παράθυρο διαλόγου που εμφανίζεται όπως παρουσιάζεται στην Εικόνα 6.10 παρέχονται αυτόματα όλες οι απαιτούμενες πληροφορίες.



Εικόνα 6.10: Παράθυρο διαλόγου αποθήκευσης αποσυμπιεσμένου εκτελέσιμου

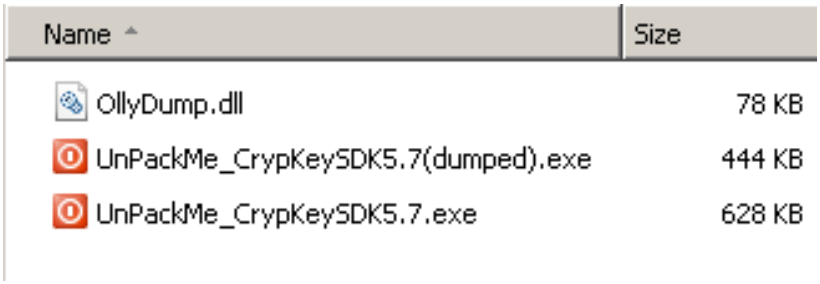
Ειδικότερα ήδη έχει συμπληρωθεί η σωστή τιμή για το πεδίο ImageBase, το μέγεθος, το πεδίο Base of Code ενώ ενημερώνει και για την αλλαγή του σημείου εισόδου από 6B6DE σε 271B0 (βλ. 3.2.2) στο οποίο αν προστεθεί το ImageBase αντιστοιχεί στην διεύθυνση που βρήκαμε προηγουμένως να είναι το αυθεντικό σημείο εισόδου (004271B0). Ακόμα μπορεί να διορθώσει τις τιμές μετατόπισης για κάθε τομέα και τα μεγέθη τους ενώ μπορεί να διορθώσει ακόμα και τις εισαγόμενες συναρτήσεις από δυναμικές βιβλιοθήκες (βλ. 3.2.4).

Η εφαρμογή που χρησιμοποιήθηκε για να συμπιέσει το αρχείο και να το προστατέψει από την αντιστροφή της δεν επηρέασε όμως τις βιβλιοθήκες. Αντίθετα μια

εφαρμογή «Protector» είναι πολύ πιθανό να αναλάμβανε να εισάγει η ίδια τις βιβλιοθήκες στο εκτελέσιμο αρχείο. Σε εκείνη την περίπτωση ο μηχανικός λογισμικού θα έπρεπε να διορθώσει χειροκίνητα τυχόν προβλήματα που δε θα μπορούσε να διορθώσει αυτόματα η πρόσθετη βιβλιοθήκη «OllyDump.dll», δηλαδή θα έπρεπε να επαναδημιουργήσει τον τομέα «.idata» της προαιρετικής κεφαλής του ΦΕ. Επομένως στο συγκεκριμένο παράδειγμα πρέπει να μην επιλεγεί η επαναδημιουργία των εισαγόμενων συναρτήσεων (Rebuild Imports).

Ένα άλλο σημείο ενδιαφέροντος είναι η ονομασία των τομέων που προστέθηκαν στο εκτελέσιμο αρχείο από την εφαρμογή CrypKey 5.7. Γενικά οι τομείς αυτοί χρησιμοποιούν τις τυπικές ονομασίες όπως «.text», «.rdata» κ.ο.κ. όμως είναι στην διακριτική ευχέρεια του εκάστοτε προγραμματιστή να τους αλλάξει ονομασία.

Πλέον το μόνο που απομένει για να αποθηκευθεί το αρχείο είναι να πατηθεί το κουμπί «Dump» και να επιλεγεί ένας προορισμός για το καινούριο αρχείο. Το νέο αρχείο όμως παρατηρεί κανείς πως είναι μικρότερο σε μέγεθος από το αρχικό όπως παρουσιάζεται στην Εικόνα 6.11. Αυτό συμβαίνει λόγω του γεγονότος πως οι δημιουργεί εφαρμογών συμπίεσης εκτελέσιμων (packers) έχουν αποκλίνει από τον αρχικό τους στόχο που είναι η μείωση του μεγέθους του εκτελέσιμου και στοχεύουν περισσότερο στο να αποτρέψουν τους μηχανικούς λογισμικού από το να αντιστρέφουν τις εφαρμογές τους. Έτσι ακόμα και μια πολύ απλή εφαρμογή συμπίεσης καταλήγει να έχει περισσότερο κώδικα αποτροπής των τεχνικών της ΑΜΛ και να προσθέτει όγκο στο εκτελέσιμο αρχείο.



Εικόνα 6.11: Σύγκριση συμπιεσμένου αρχείου και αποσυμπιεσμένου

Τέλος, εκτελώντας το νέο εκτελέσιμο αρχείο το λειτουργικό σύστημα δεν έχει κανένα πρόβλημα και εμφανίζεται το μήνυμα της εφαρμογής κανονικά.

6.4 Άλλες τεχνικές προστασίας.

Σε πραγματικές εφαρμογές τα πράγματα δεν είναι σχεδόν ποτέ τόσο απλά καθώς οι προγραμματιστές έχουν εφεύρει πάρα πολλές μεθοδολογίες ώστε να μπορέσουν να προστατέψουν τον κώδικα τους. Σε αυτήν την ενότητα θα γίνει μια προσπάθεια να παρουσιαστεί η λογική πίσω από μερικές από τις πιο κοινές τεχνικές που μπορεί να συναντήσει κάποιος στην προσπάθεια του να αντιστρέψει εμπορικές εφαρμογές και όχι να παρουσιαστεί η διαδικασία αντιστροφής των λειτουργιών αυτών.

Ο στόχος δεν είναι άλλος από το να μπορέσει ο μηχανικός λογισμικού να κατανοήσει καλύτερα τα πολλά επίπεδα στα οποία μπορεί ο προγραμματιστής να εστιάσει για να προστατέψει την εφαρμογή του καθώς και ο απλός προγραμματιστής να πάρει ιδέες που θα τον βοηθήσουν να δημιουργήσει ποιοτικότερο και ασφαλέστερο κώδικα.

6.4.1 Έλεγχοι κυκλικού κώδικα πλεονασμού (CRC Checks).

Σίγουρα πολλές φορές κατά την ενασχόληση με αρχεία υπολογιστών έχει εμφανιστεί το μήνυμα σφάλματος ελέγχου κυκλικού πλεονασμού είτε πρόκειται για μια αντιγραφή αρχείων από δίσκο σε δίσκο ή για αποσυμπίεση αρχείων. Αυτό συμβαίνει όταν το αρχικό αρχείο διαφέρει από το αρχείο προορισμού. Πολλές φορές επίσης όταν μεταφορτώνεται κάποιο αρχείο από το διαδίκτυο στον υπολογιστή συνοδεύεται με έναν οχταψήφιο δεκαεξαδικό αριθμό ο οποίος χρησιμοποιείται για την επαλήθευση της επιτυχούς μεταφόρτωσης του αρχείο. Ο αριθμός ονομάζεται κυκλικώς κώδικας πλεονασμού (CRC). Υλοποιήσεις του ελέγχου αυτού συναντάμε τόσο σε εκτελέσιμα αρχεία όσο και στα δίκτυα υπολογιστών για την πιστοποίηση της αυθεντικότητας των πακέτων. Ο τρόπος λειτουργίας του αλγορίθμου ελέγχου κυκλικού κώδικα πλεονασμού είναι κοινός.

Ένα κοινό λάθος που συμβαίνει είναι να μεταφράζεται ο όρος CRC σαν Cyclic Redundancy Check δηλαδή έλεγχος κυκλικού πλεονασμού αλλά και να χρησιμοποιείται λάθος. Έτσι πολλές φορές χρησιμοποιείται η φράση «έλεγχος CRC» με την λάθος μετάφραση του όρου χωρίς να βγάζει νόημα. Αντιθέτως όμως με τη σωστή απόδοση του

όρου CRC ως Cyclic Redundancy Code, δηλαδή κώδικας κυκλικού πλεονασμού μπορεί να χρησιμοποιηθεί η έκφραση «έλεγχος CRC» ή «CRC Check».

Όσον αφορά τον τρόπο υπολογισμού του CRC είναι αρκετό για τον μηχανικό λογισμικού να γνωρίζει πως αποτελεί το υπόλοιπο της δυαδικής διαίρεσης του δυαδικού πολυωνύμου που προκύπτει από την δυαδική πληροφορία του αρχείου, προς ένα δεύτερο δυαδικό πολυώνυμο (πολυώνυμα γεννήτορας). Σε κάθε διαίρεση θα υπάρχει ένα υπόλοιπο το οποίο θα είναι τουλάχιστον μικρότερο κατά ένα bit από το αρχικό πολυώνυμο και μπορεί να ισούται και με το 0. Σαν αποτέλεσμα όμως όταν το αρχείο τροποποιηθεί και διαιρεθεί με το ίδιο πολυώνυμο διαιρέτη, το υπόλοιπο θα διαφέρει, με αποτέλεσμα να μπορεί να επαληθευθεί η αυθεντικότητα του αρχείου. Βέβαια υπάρχει μια πιθανότητα να γίνει λάθος επαλήθευση αλλά αυτή αντιστοιχεί με πιθανότητα $1/2^{32}$ όταν χρησιμοποιείται έλεγχος CRC των 32bit στην οποία περίπτωση ο διαιρέτης αποτελείται από 32bit.

Εύλογο είναι λοιπόν πως οι προγραμματιστές για να προστατέψουν τα αρχεία τους οδηγήθηκαν στη σκέψη πως ένας έλεγχος CRC θα είχε μεγάλες πιθανότητες να μπορεί να αποτρέψει κάθε παρέμβαση στα εκτελέσιμα αρχεία.

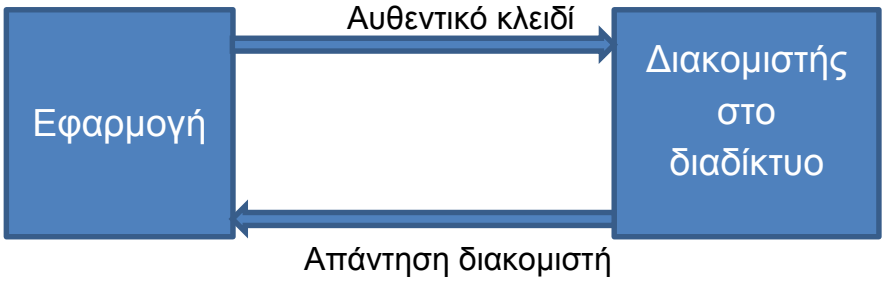
Επομένως ο μηχανικός λογισμικού σε περίπτωση που θέλει να πραγματοποιήσει αλλαγές σε αυτά τα αρχεία θα πρέπει να ανακαλύψει τον αλγόριθμο ελέγχου CRC και να τροποποιήσει την τιμή CRC. Ακόμα πολλές φορές αυτή η τιμή μπορεί να έχει τεθεί στατικά στο εκτελέσιμο αρχείο και να πρέπει να διορθωθεί και εκεί. Ακόμα μπορεί να γίνει επιδιόρθωση του κώδικα ώστε να μη γίνεται ποτέ ο έλεγχος προσπερνώντας τις αλλαγές ροής που οδηγούν στα μηνύματα αποτυχίας. Τέλος υπάρχουν στην ΑΜΛ έχουν δημιουργηθεί εφαρμογές οι οποίες απαλλάσσουν τον μηχανικό από κάθε σχέση με τους ελέγχους αυτούς καθώς υπολογίζουν την τιμή του CRC και διορθώνουν αυτόματα τους ελέγχους CRC με τη νέα τιμή.

6.4.2 Έλεγχοι μέσω διακομιστών (Server Checks).

Μια δεύτερη τεχνική που έχει ανθίσει με την ευρεία είσοδο των χρηστών πληροφορικής στο ευρυζωνικό διαδίκτυο όπου σχεδόν κάθε σπίτι έχει αποκτήσει και μια σύνδεση στο διαδίκτυο και πόσο μάλλον για τους εργαζόμενους στον χώρο της πληροφορικής που έχει γίνει μια αναγκαιότητα, είναι η πραγματοποίηση αυθεντικότητας

των χρηστών ή των αντιτύπων των εφαρμογών που «υποτίθεται» έχουν αγοράσει νόμιμα οι χρήστες σε διαδικτυακούς διακομιστές και όχι απαραίτητα μέσα στο εκτελέσιμο αρχείο.

Συνήθως παρατηρείται σε εφαρμογές που χρησιμοποιούν ελέγχους με κλειδάριθμους ή σειριακούς αριθμούς ώστε να μην είναι εφικτή η χρήση γεννητριών κλειδιών για τη δημιουργία αυθεντικών κλειδιών. Η διαδικασία βασίζεται στη λογική πως ακόμα και αν ο χρήστης διαθέτει ένα κλειδί το οποίο μοιάζει αυθεντικό, αν το κλειδί δεν έχει κυκλοφορήσει στην αγορά από την εταιρεία παύει να είναι αυθεντικό. Επομένως η εταιρεία πλέον κρατάει στοιχεία σε μια βάση δεδομένων στον διαδικτυακό διακομιστή της με όλα τα κλειδιά που έχουν δοθεί στην κυκλοφορία. Η εφαρμογή στη συνέχεια εφόσον πιστοποιήσει πως ένα κλειδί είναι αυθεντικό προσπαθεί να επικοινωνήσει με τον διακομιστή για να ελέγξει για δεύτερη φορά την αυθεντικότητα του κλειδιού και να πιστοποιήσει την ύπαρξη του στην βάση δεδομένων του διακομιστή. Στη συνέχεια ο διακομιστής αναλόγως με τα αποτελέσματα του δικού του ελέγχου «απαντά» στην εφαρμογή με έναν τρόπο που έχει καθοριστεί κατά την δημιουργία της εφαρμογής και έτσι την έχει ενημερώσει για το αν είναι αυθεντικό το κλειδί που εισήχθη.



Εικόνα 6.12: Έλεγχος διακομιστή

Το πλεονέκτημα αυτής της μεθόδου είναι πως καθίσταται κάθε γεννήτρια κλειδιών σχεδόν άχρηστη λόγω της αλληλεπίδρασης με τον διακομιστή στο διαδίκτυο ο οποίος ελέγχει τη βάση δεδομένων του. Ακόμα ο μηχανικός λογισμικού δεν έχει πρόσβαση στους μηχανισμούς του διακομιστή και έτσι δεν μπορεί να τον επηρεάσει. Το μειονέκτημα της μεθόδου αυτής είναι πως αναγκάζει τον χρήστη να έχει πρόσβαση στο διαδίκτυο με αποτέλεσμα να περιορίζει το αγοραστικό κοινό της εφαρμογής και να δίνει μια αίσθηση ανασφάλειας στον χρήστη. Ακόμα μπορεί να παρουσιάσει προβλήματα με αντι-ιικό

λογισμικό και τείχη προστασίας (firewalls) τα οποία αντιμετωπίζουν τέτοιες συμπεριφορές εφαρμογών με «καχυποψία».

Στην ΑΜΛ τέτοιες συμπεριφορές αντιμετωπίζονται πάλι με παρόμοιες τεχνικές. Αρχικά μπορεί ο μηχανικός λογισμικού να επιδιορθώσει το τμήμα κώδικα στο οποίο γίνεται η κλήση στον διακομιστή ώστε να το αποτρέψει από το να συμβεί. Επίσης μπορεί να αντιστρέψει τον αλγόριθμο που ελέγχει την απάντηση του διακομιστή και να επέμβει σε αυτό το σημείο ώστε να επικυρώνεται πάντα η απάντηση του διακομιστή.

6.4.3 Κρυπτογράφηση ονομάτων – συμβολοσειρών (Obfuscation).

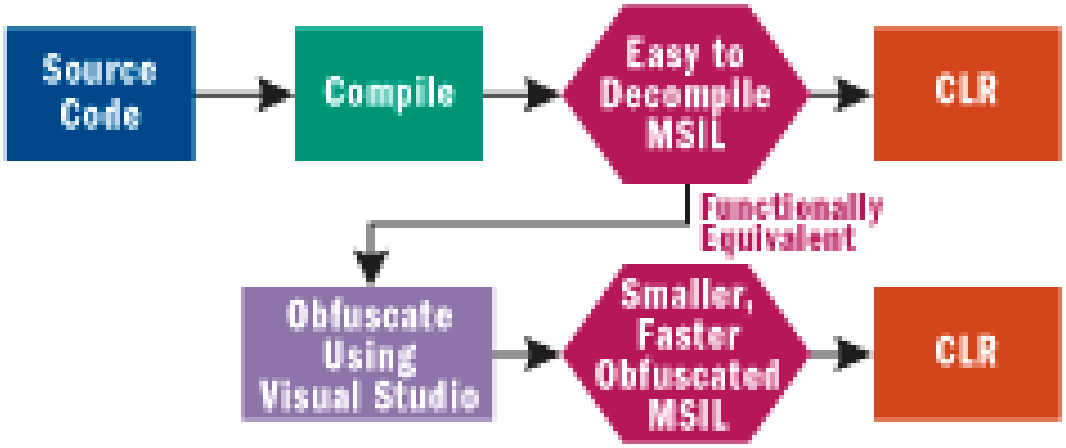
Η τεχνική της κρυπτογράφησης ονομάτων και συμβολοσειρών αφορά κυρίως εφαρμογές γραμμένες σε γλώσσες που βασίζονται σε κάποιο framework για να λειτουργήσουν. Χαρακτηριστικά παραδείγματα είναι το .NET Framework και η Java. Σε αυτές τις γλώσσες όπως παρουσιάστηκε στην ενότητα 4.2 είναι δυνατή η πλήρης απομεταγλώττιση τους με αποτέλεσμα να καθίσταται πολύ εύκολη η τροποποίηση τους από τον μηχανικό λογισμικού καθώς ο κώδικας είναι απόλυτα κατανοητός.

Επομένως από τη στιγμή που μπορεί ο χρήστης να κατανοήσει κάθε σημείο του κώδικα εύλογο είναι πως σχεδόν κάθε σημείο θα μπορεί να τροποποιηθεί ώστε να αποφευχθεί κάθε προσπάθεια προστασίας της εφαρμογής που έχει καταβάλλει ο προγραμματιστής. Το τεράστιο αυτό κενό ασφαλείας που δημιουργείται από την φύση αυτών των γλωσσών προγραμματισμού δημιούργησε την ανάγκη να πρέπει με κάποιον τρόπο να αποκρύψουν οι προγραμματιστές τον κώδικα τους ώστε να μην είναι τόσο εύκολα αναγνώσιμος.

Αν το σκεφτεί κανείς καλύτερα, τα στοιχεία που κάνουν αναγνώσιμο και κατανοητό τον κώδικα σε μια γλώσσα υψηλού επιπέδου αναμφίβολα είναι οι συμβολικές ονομασίες που μπορεί να δώσει ο προγραμματιστής σε μεθόδους, συναρτήσεις, μεταβλητές και συμβολοσειρές. Σίγουρα μια δήλωση στην C# του τύπου «string username;» είναι πολύ πιο κατανοητή από μια δήλωση του τύπου «string abcdefg;» όσον αφορά την λειτουργικότητα της μεταβλητής όμως αυτό βοηθάει αποκλειστικά τον προγραμματιστή και όχι τον Η/Υ για τον οποίο οι δύο δηλώσεις σε γλώσσα μηχανής επιτελούν την ίδια λειτουργικότητα της ανάθεσης ενός χώρου στην μνήμη για την αποθήκευση μιας

συμβολοσειράς. Το ίδιο συμβαίνει για την ονομασία κάθε μεθόδου ή συνάρτησης και για κάθε συμβολοσειρά.

Αυτή η διαπίστωση οδήγησε στην δημιουργία της τεχνικής της απόκρυψης συμβολοσειρών (obfuscation) η οποία με τη χρήση αλγόριθμων κρυπτογράφησης εξελίχτηκε αρκετά ώστε να χρησιμοποιούνται εφαρμογές που αποκρύπτουν τα ονόματα και όλες τις συμβολοσειρές του πηγαίου κώδικα κρυπτογραφώντας τες σε μεγάλες συμβολοσειρές αλφαρηθμιτικών. Το αποτέλεσμα είναι μια εφαρμογή ταυτόσιμη με την αρχική σε λειτουργικότητα αλλά πολύ δύσκολα αναγνώσιμης από τον μηχανικό λογισμικού. Το σημαντικό στοιχείο όμως είναι πως αυτή η διαδικασία της απόκρυψης δεν συμβαίνει στον πηγαίο κώδικα όπως τον διαβάζει ο προγραμματιστής που δημιουργεί την εφαρμογή αλλά συμβαίνει μετά την μεταγλώττιση όπως παρουσιάζεται στην Εικόνα 6.13. Επομένως ο προγραμματιστής που έχει στα χέρια του τον πηγαίο κώδικα δεν ταλαιπωρείται από αυτήν την διαδικασία αλλά επηρεάζεται μόνο η διαδικασία της αντιστροφής της εφαρμογής.



Εικόνα 6.13: Η διαδικασία της κρυπτογράφησης σε .NET εφαρμογές (Gabriel Torok & Bill Leach, 2003)

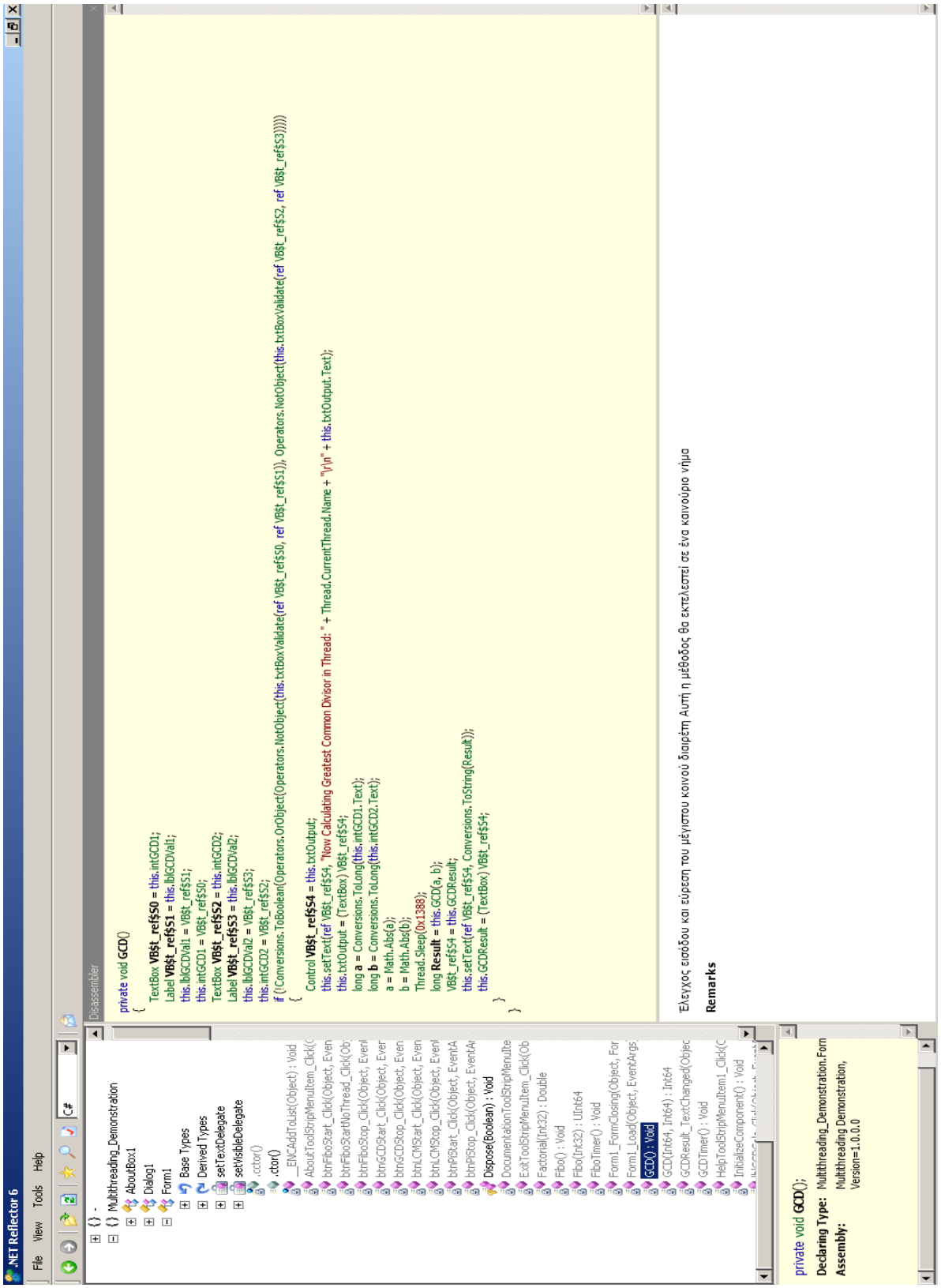
Ακόμα με την μέθοδο αυτή μπορεί να στοχευθεί πληροφορία που αποθηκεύεται σε πολλά σημεία των μεταγλωττισμένων αρχείων με τεχνικές όπως:

1. Η μετονομασία των μεταδεδομένων του αρχείου.
2. Αφαίρεση μη απαραίτητων μεταδεδομένων.
3. Κρυπτογράφηση συμβολοσειρών.

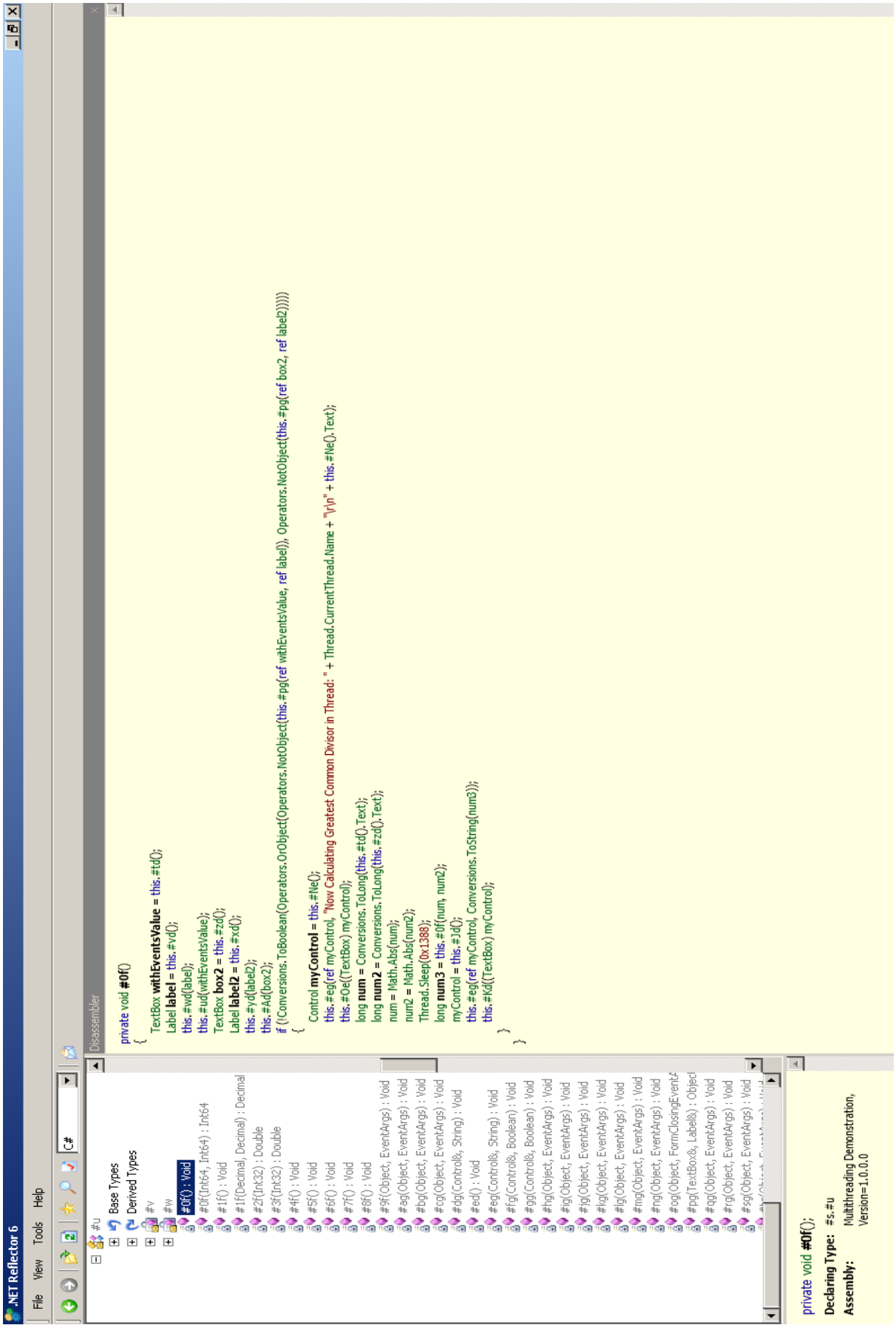
Τα εργαλεία που χρησιμοποιούνται για την απομεταγλώττιση τέτοιων εφαρμογών λόγω της έλλειψης μεταδεδομένων συνήθως εμφανίζουν μηνύματα αποτυχίας κατά την

απομεταγλώττιση. Ακόμα όμως και αν καταφέρουν να επιτύχουν την απομεταγλώττιση σε καμία περίπτωση ο κώδικας δε θα είναι ευανάγνωστος. Στην Εικόνα 6.14 και στην Εικόνα 6.15 παρουσιάζεται η διαφορά μεταξύ ενός μη κρυπτογραφημένου, απομεταγλωττισμένου τμήματος κώδικα μιας εφαρμογής που υπολογίζει τον μέγιστο κοινό διαιρέτη και της ίδιας εφαρμογής κρυπτογραφημένης. Η κρυπτογράφηση έγινε με τη δοκιμαστική έκδοση της εφαρμογής Smart Assembly v6.2 της εταιρείας «Red Gate» (http://www.red-gate.com/products/dotnet-development/smartassembly/download). Όπως βλέπει κανείς τα ονόματα όλων των μεθόδων έχουν αλλάξει σε συμβολοσειρές ASCII. Το ίδιο έχει συμβεί και στα ονόματα μεταβλητών. Ακόμα αν κάποιος μελετήσει περισσότερο τα αποτελέσματα θα δει μετονομασίες σε πολλές συμβολοσειρές και επίσης την προσθήκη περισσότερου κώδικα στην εφαρμογή. Σαν αποτέλεσμα η πολυπλοκότητα της αναγνώρισης των βασικών δομών μέσα στην εφαρμογή αυξήθηκε σε μεγάλο βαθμό. Η εφαρμογή Smart Assembly έχει ακόμα και τη δυνατότητα χρήσης συμβόλων και μη αναγνώσιμων χαρακτήρων για τις μετονομασίες αλλά και πολύ πιο σύνθετες δυνατότητες όπως την υποστήριξη ίδιων ονομάτων για διαφορετικές μεθόδους και μεταβλητές που μπορεί να δημιουργήσουν τεράστια σύγχυση κατά την αντιστροφή της εφαρμογής.

Όπως συμπεραίνει κανείς η κρυπτογράφηση αποτελεί την βασικότερη γραμμή άμυνας για τον προγραμματιστή που ασχολείται με γλώσσες βασισμένες σε Frameworks αλλά και πάλι από τη στιγμή που επιτρέπει στον μηχανικό λογισμικού να την απομεταγγλωτίσει είναι θέμα χρόνου μέχρι να κατανοήσει τη δομή της εφαρμογής και να μπορέσει να επέμβει στις λειτουργίες της. Θα μπορούσε κανείς να πει πως ενώ δυσχεραίνει τη δουλειά του μηχανικού λογισμικού στην πραγματικότητα δεν μπορεί να προστατέψει την εφαρμογή αποτελεσματικά.



Εικόνα 6.14: Απομεταγλωττισμένη εφαρμογή υπολογισμού του μέγιστου κοινού διαιρέτη χωρίς κρυπτογράφηση



Εικόνα 6.15: Η εφαρμογή της εικόνας 6.14 μετά την κρυπτογράφηση της εφαρμογής προστασίας Smart Assembly 6.2

6.5 Σύνοψη.

Στην ενότητα αυτή συζητήθηκαν οι βασικές μεθοδολογίες που χρησιμοποιούνται για την προστασία εφαρμογών από τις τεχνικές της ΑΜΛ τόσο με τη χρήση κώδικα προστασίας από τον προγραμματιστή όσο και με την αγορά εφαρμογών προστασίας.

Συγκεκριμένα παρουσιάστηκε μέσα από απλά παραδείγματα η αποτροπή από την αποσφαλμάτωση στοχεύοντας τα εργαλεία αποσφαλμάτωσης και ειδικότερα ελέγχοντας την ύπαρξη τους κατά την εκτέλεση της εφαρμογής. Επίσης παρουσιάστηκε πόσο αναποτελεσματικός είναι αυτός ο τρόπος να προστατέψει την εφαρμογή από έμπειρους μηχανικούς λογισμικού. Ακόμα παρουσιάστηκαν εφαρμογές που σχεδιάστηκαν για να προστατέψουν τα εκτελέσιμα αρχεία χρησιμοποιώντας τεχνικές όπως η συμπίεση του εκτελέσιμου αρχείου (packing). Τέλος έγινε μια αναφορά σε άλλες τεχνικές όπως οι εφαρμογές «protectors» που δουλεύουν λίγο πιο σύνθετα από τις εφαρμογές «packers» και όχι μόνο δε μειώνουν το μέγεθος των εκτελέσιμων αρχείων αλλά στην προσπάθεια τους να παρέχουν ολοκληρωτική προστασία πολλαπλασιάζουν το μέγεθος των αρχείων.

Στη συνέχεια έγινε αναφορά σε τεχνικές που χρησιμοποιούνται ευρέως σε εμπορικές εφαρμογές όπως οι έλεγχοι κώδικα κυκλικού πλεονασμού, οι έλεγχοι μέσω διακομιστών διαδικτύου. Τέλος παρουσιάστηκε η βασική μεθοδολογία προστασίας εφαρμογών γραμμένων σε γλώσσες προγραμματισμού βασισμένες στο .NET Framework η οποία έχει εφαρμογές και σε άλλες γλώσσες όπως η Java.

Στο επόμενο κεφάλαιο θα παρουσιαστούν κάποια βασικά συμπεράσματα τα οποία προκύπτουν από τη μέχρι τώρα αποκτηθείσα γνώση τόσο για τα πρακτικά ζητήματα όσο και για τα ηθικά ζητήματα που προκύπτουν.

Συμπεράσματα – Προτάσεις

Μελετώντας τους διάφορους τρόπους προστασίας ενός φορητού εκτελέσιμου αρχείου που παρουσιάστηκαν στην εργασία και στην συνέχεια αντιστρέφοντας τους καταλήγουμε στο γενικό συμπέρασμα πως καμία μορφή προστασίας δεν μπορεί να προσφέρει σε απόλυτο βαθμό αποτροπή της αντιστροφής της λειτουργίας του εκτελέσιμου καθώς εφόσον ο μηχανικός λογισμικού κατέχει τις απαραίτητες γνώσεις και διαθέτει αρκετό χρόνο μπορεί να την παρακάμψει. Παρόλα αυτά όμως οι διάφορες τεχνικές που αναπτύχθηκαν σταδιακά στην προσπάθεια αυτή βρίσκουν καθημερινή εφαρμογή σε εμπορικές εφαρμογές για να προστατέψουν την πνευματική ιδιοκτησία των δημιουργών τους.

Παρότι ο στόχος τους είναι θεωρητικά ανέφικτος, έχουν καταφέρει σε μεγάλο βαθμό να δυσχεραίνουν το έργο των μηχανικών λογισμικού και να τους καθυστερούν σε πολύ μεγάλο βαθμό. Έτσι διαφυλάσσονται για αρκετό καιρό τα πνευματικά δικαιώματα της εταιρείας. Ειδικότερα στις βιομηχανίες παιχνιδιών η καθυστέρηση του «σπασίματος» μιας εφαρμογής οδηγεί πάρα πολλούς να αγοράσουν το αυθεντικό λογισμικό όσο είναι ακόμα επίκαιρο γιατί σίγουρα κάθε παιδί θα ήθελε να παίζει ένα παιχνίδι την ημέρα που βγαίνει στην κυκλοφορία αν αυτό είναι δυνατόν. Σαν αποτέλεσμα κάθε μέρα που ο διαρρήκτης λογισμικού καθυστερεί η εταιρεία αυξάνει το κέρδος της γιατί ο πελάτης αναγκάζεται να αγοράζει το λογισμικό.

Το ερώτημα που προκύπτει είναι ποιο είναι το όριο στο οποίο πρέπει οι δημιουργοί εφαρμογών να προστατεύουν τις εφαρμογές τους. Εξάλλου εύλογο είναι πως όσο περισσότερες και πιο σύνθετες τεχνικές προστασίας υιοθετούνται τόσο αυξάνεται το κόστος παραγωγής του λογισμικού και οι εργατοώρες που απαιτούνται για την ολοκλήρωση του έργου. Σίγουρα δεν είναι εύκολο να απαντηθεί το ερώτημα αλλά μπορεί να γίνει μια κριτική των διαφόρων τεχνικών που χρησιμοποιήθηκαν ώστε ο προγραμματιστής να αποκτήσει καλύτερη αντίληψη της πορείας που πρέπει να ακολουθήσει. Οι τεχνικές που μελετήθηκαν είναι οι εξής:

1. **Η αύξηση της πολυπλοκότητας του αλγορίθμου ταυτοποίησης του χρήστη ή του αυθεντικού κλειδιού:** Αυτή η λύση αναμφίβολα προκάλεσε τη

μεγαλύτερη καθυστέρηση στην ανάλυση της εφαρμογής. Ο μηχανικός λογισμικού αναγκάζεται με αυτόν τον τρόπο να αναλύσει πολύ περισσότερο όγκο κώδικα και έτσι καθυστερεί να βρει λύση όμως αυτή η τεχνική από μόνη της είναι ανεπαρκής γιατί ακόμα και ο πιο άπειρος μηχανικός λογισμικού μπορεί να επενδύσει χρόνο σε τέτοιου είδους προστασίες και να τα καταφέρει. Ακόμα η εμπειρία μπορεί να τον βοηθήσει να αναγνωρίζει τα ασήμαντα σημεία κώδικα πιο γρήγορα και να επιταχύνει τη διαδικασία.

2. **Αντι-αποσφαλμάτωση:** Πολύ απλός τρόπος για να προστατευτεί μια εφαρμογή από άπειρους μηχανικούς λογισμικού ο οποίος πρακτικά δεν κοστίζει ιδιαίτερα στην εφαρμογή. Το μόνο μειονέκτημα προκύπτει από το γεγονός πως καθυστερεί η αρχική φόρτωση της εφαρμογής και την απλούστατη μέθοδο παράκαμψης της.
3. **Εφαρμογές «packers» και «protectors»:** Πρόκειται για μια δραστική λύση η οποία έχει θετικά και αρνητικά στοιχεία. Η συμπίεση των εκτελέσιμων αρχείων προσφέρει σημαντικά πλεονεκτήματα από τη μείωση του όγκου αλλά καθόλου ουσιαστική προστασία από έμπειρους μηχανικούς λογισμικού. Μπορεί όμως να αποτρέψει άπειρους μηχανικού λογισμικού οι οποίοι δεν έχουν τις γνώσεις να αποσυμπιέσουν το εκτελέσιμο αρχείο για να μπορέσουν να επέμβουν. Αντίθετα οι εφαρμογές «protectors» καταργούν όλα τα πλεονεκτήματα της συμπίεσης των εκτελέσιμων αρχείων και δημιουργούν πιο απαιτητικές εφαρμογές με προβλήματα συμβατότητας με αντι-ιικό λογισμικό. Προσφέρουν πολύ καλή προστασία στις εφαρμογές μέχρι να γίνει η πρώτη αντιστροφή της συγκεκριμένης έκδοσης της εφαρμογής προστασίας οπότε και κάθε εκτελέσιμο που την χρησιμοποιεί χάνει την προστασία του. Αυτό το πρόβλημα προκύπτει και για εφαρμογές «packers» όσο και για «protectors».
4. **Κρυπτογράφηση ονομάτων– συμβολοσειρών**: Αποτελεί μονόδρομο για κάθε εφαρμογή γραμμένη σε γλώσσες που βασίζονται σε Frameworks. Δεν νοείται εφαρμογή η οποία γράφεται σε τέτοιες γλώσσες και δεν χρησιμοποιεί αυτήν την τεχνική προστασίας ως το ελάχιστο δυνατό. Παράλληλα δεν δημιουργεί προβλήματα και καθυστερήσεις στην εκτέλεση της εφαρμογής.
5. **Έλεγχοι κώδικα κυκλικού πλεονασμού**: Παρακάμπτονται τόσο εύκολα που και αποσκοπούν μόνο στην αποτροπή άπειρων μηχανικών λογισμικού.

6. **Έλεγχοι μέσω διακομιστών**: Πρόκειται για μια πολύ καλή λύση όταν χρησιμοποιείται για εφαρμογές που ούτως ή άλλως απαιτούν την ύπαρξη διαδικτύου αλλιώς προκαλούν μείωση του αγοραστικού κοινού στο οποίο απευθύνεται η εφαρμογή.

Έτσι μπορεί κανείς να πει πως η επιλογή της τεχνικής προστασίας είναι κάτι που επιλέγεται με βάση τον προϋπολογισμό του έργου ανάπτυξης μιας εφαρμογής λογισμικού και του κέρδους που προβλέπεται από τις πωλήσεις της. Σίγουρα όμως οι τεχνικές που είναι απλές και δεν επιβαρύνουν την εφαρμογή θα πρέπει να χρησιμοποιούνται σε όλες τις εφαρμογές ώστε να μην κινδυνεύουν από όλους τους μηχανικούς λογισμικού αλλά μόνο από άτομα με εμπειρία στην ΑΜΛ. Ακόμα σε περίπτωση που χρειάζεται μια βαθμίδα προστασίας παραπάνω αξίζει να επενδυθεί χρόνος για τη δημιουργία μιας τροποποιημένης εφαρμογής συμπίεσης εκτελέσιμων αρχείων η οποία δε θα είναι διαθέσιμη στο εμπόριο. Έτσι θα είναι πιο απίθανο να έχει ασχοληθεί κάποιος από τους κορυφαίους της υπόγειας σκηνής ώστε να παράγει εργαλεία για την ευκολότερη παράκαμψη της συγκεκριμένης προστασίας ενώ το εκτελέσιμο αρχείο θα έχει και όλα τα πλεονεκτήματα που αναφέρθηκαν προηγουμένως.

Καταλήγοντας όμως πρέπει να γίνει και μια αναφορά σε ένα μεγάλο ηθικό ζήτημα που προκύπτει και αφορά την θεμιτή χρησιμοποίηση της επιστήμης της ΑΜΛ που αφορά στη συντήρηση και στην βελτίωση εφαρμογών που έχουν αγοραστεί νόμιμα. Αυτό το ζήτημα εξακολουθεί να παραμένει διφορούμενο και οι απόψεις διίστανται καθώς στην προσπάθεια να προστατευτούν τα πνευματικά δικαιώματα από τους κακοπροαίρετους χρήστες της ΑΜΛ επηρεάζονται και οι χρήστες που επιθυμούν να την χρησιμοποιήσουν θεμιτά. Αν όλες οι εφαρμογές ήταν αδύνατον να αντιστραφούν τότε δε θα υπήρχε περιθώριο επιδιόρθωσης προβληματικών εφαρμογών του παρελθόντος και βελτίωσης των. Ακόμα άτομα που χρησιμοποιούν εφαρμογές εταιρειών που έχουν κλείσει θα ήταν αδύνατον να διορθώσουν τα σφάλματα σχεδίασης τους. Επομένως ακόμα και στην δίκαιη προσπάθεια προστασίας των πνευματικών δικαιωμάτων πρέπει να υπάρχει και κάποιο όριο.

Βιβλιογραφία - Αναφορές

Abd-El-Barr, M., & El-Rewini, H. (2005). *Fundamentals of Computer Organization and Architecture.* New Jersey: Wiley Publishing, Inc.

Backer Street Software. (2007). *REC Decompiler Home Page*. Ανάκτηση Αύγουστος 11, 2010, από REC Decompiler: http://www.backerstreet.com/rec/rec.htm

Blum, R. (2005). *Professional Assembly Language.* Wiley Publishing, Inc.

CDKiLLER, T. (n.d.). Ανάκτηση Αύγουστος 3, 2011, από Protection ID: http://pid.gamecopyworld.com/ProtectionID_v6.4.0.rar

Eilam, E. (2005). *Reversing - Secrets of Reverse Engineering.* Indianapolis: Wiley Publishing, Inc.

http://msdn.microsoft.com/en-us/magazine/cc164058.aspx#S3. (2003, Νοέμβριος). *MSDN Magazine*. Ανάκτηση Οκτωβρίου 5, 2011, από MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/cc164058.aspx#S3

Hyde, R. (2006). *WRITE GREAT CODE, Vol. 2: Thinking Low-Level, Writing High-Level.* San Francisco: No Starch Press.

Kaminsky, D., Ferguson, D., Larsen, J., Miras, L., & Pearce, W. (2008). *Reverse Engineering Code with IDA Pro.* USA: Elsevier Inc.

Lena. (n.d.). *Tuts4You.* Ανάκτηση Σεπτέμβριος 15, 2011, από Tuts4You: http://tuts4you.com/download.php?view.141

Microsoft. (NA). *Microsoft Portable Executable and Common Object File Format Specification.* Ανάκτηση Αύγουστος 1, 2010, από Microsoft: http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx

Microsoft. (n.d.). *Win32 API Reference.* Ανάκτηση Σεπτέμβριος 15, 2011, από NA: http://win32assembly.online.fr/files/win32api.zip

NA. (NA, NA NA). *x86 Registers*. Ανάκτηση Ιούλιος 31, 2010, από x86 Registers: http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html

Perry, M., & Oskov, N. (2003). *Introduction To Software Reverse Engineering.* NA: NA.

Pietrek, M. (2002, Φεβρουάριος). An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*.

Plachy, J. (1996-1997). *The Portable Executable File Format.* Ανάκτηση Αύγουστος 1, 2010, από Tuts4You: http://tuts4you.com/download.php?view.2892

Systa, T. (2000). *Static And Dynamic Reverse Engineering Techniques For Java Software Sysytems.* Tampere: Static And Dynamic Reverse Engineering Techniques For Java Software Sysytems.

Αλεξόπουλος, Ά., & Λαγογιάννης, Γ. (2003). *Τηλεπικοινωνίες Και Δίκτυα Υπολογιστών Έκτη Έκδοση.* Αθήνα: ΑΦΟΙ ΡΟΗ Α.Ε.

NA. (n.d.). *OpenRCE.* Ανάκτηση Σεπτεμβρίου 20, 2011, από OpenRCE: http://www.openrce.org/downloads/download_file/108

Παράρτημα Α

Εφαρμογή SimpleFor.cpp

Γλώσσα: C++

Προγραμματιστής: Τζιώτζης Γαβριήλ

Σχόλια:

Πηγαίος κώδικας:

```
int main()
{
    int a=0;
    for (int i=0; i<10; i++)
    {
        a++;
    }
    return 0;
};
```

Εφαρμογή SimpleIF.cpp

Γλώσσα: C++

Προγραμματιστής: Τζιώτζης Γαβριήλ

Σχόλια:

Πηγαίος κώδικας:

```
#include <string>
#include <iostream>

using namespace std;

int main()
{

    //Declaring the variables that hold login information for the authenticated user of the system.
    string authenticatedUser="admin";
    string authenticatedPassword="adminPassword";

    //Declaring the variables that will be used as containers for the information entered by the user.
    string user;
    string password;



    //Requesting and reading username
    cout<<"Please enter your username"<<endl;
    cin>>user;

    //Requesting and reading password
    cout<<"Please enter your password"<<endl;
    cin>>password;
```

```
    //If username and password combination is not valid the system exits.
    if ((authenticatedUser.compare(user) == 0) && (authenticatedPassword.compare(password) == 0))
        cout<<"Welcome admin!"<<endl;
    else
        cout<<"Invalid username or password. System will now terminate!"<<endl;

    //Code to hold execution.
    cin>>user;
    return 0;
};
```

Εφαρμογή SimpleValidate.cpp

Γλώσσα: C++

Προγραμματιστής: Τζιώτζης Γαβριήλ

Σχόλια:

Πηγαίος κώδικας:

```
#include <string>
#include <iostream>

using namespace std;

static bool validate(string user, string password)
{
    if (user.length()<10)
        return false;
    else if (user.substr(2, 5).compare("admin")!=0)
    {
        cout<<user.substr(2,5)<<endl;
        return false;
    }
    else if (password.length()<15)
        return false;
    else
        return true;

};

static long factorial(int n)
{
    if (n==0)
        return 1;
```

```
    else if (n==1)
        return 1;
    else if (n>19)
    {
        cout<<"The number you entered is too big"<<endl;
        return -1;
    }
    else if (n<0)
    {
        cout<<"You can't count factorial for negative numbers"<<endl;
        return -2;
    }
    else
        return n*factorial(n-1);
};

int main()
{

    //Declaring the variables that will be used as containers for the information entered by the user.
    string user;
    string password;

    cout<<"This progam will calculate factorial based on your inpu"<<endl;
    cout<<"To use this program you must first login"<<endl<<endl;

    //Requesting and reading username
    cout<<"Please enter your username"<<endl;
    cin>>user;

    //Requesting and reading password
    cout<<"Please enter your password"<<endl;
    cin>>password;
```

```
    //If username and password combination is not valid the system exits.
    if (validate(user, password))
        cout<<"Welcome admin!"<<endl<<endl;
    else
    {
        cout<<"Invalid username or password. System will now terminate!"<<endl<<endl;
        return 1;
    }

    //Factorial calculation
    cout<<"Enter a negative number to exit"<<endl;
    long int n=-5;
    do
    {
        cout<<"Enter a number to calculate it's factorial"<<endl;
        cin>>n;
        if (!validate(user, password))
            return 2;
        if (n>=0)
            cout<<"Factorial of "<<n<<"= "<<factorial(n)<<endl<<endl;
        else
            cout<<"Thank you for using our program"<<endl;
    }
    while (n>=0);

    return 0;
};
```

Εφαρμογή SimpleValidateKeygen

Γλώσσα: Visual C#

Προγραμματιστής: Τζιώτζης Γαβριήλ

Σχόλια: Παρατίθεται μόνο το αρχείο «Form1.cs» της εφαρμογής το οποίο χειρίζεται την δημιουργία των κλειδιών.

Πηγαίος κώδικας: «Form1.cs»

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        private void button1_Click(object sender, EventArgs e)
        {
            char[] temp = new char[15];

            Random rand = new Random();
            for (int i = 0; i < 2; i++)
            {
```

```
            int randomNumber = rand.Next(33, 126);
            temp[i] = (char) randomNumber;
        }
        temp[2]='a';
        temp[3]='d';
        temp[4]='m';
        temp[5]='i';
        temp[6]='n';
        for (int i = 7; i < 10; i++)
        {
            int randomNumber = rand.Next(33, 126);
            temp[i] = (char) randomNumber;
        }
        string s = new string(temp);
        textBox1.Text = s;
        Array.Clear(temp, 0, 15);
        for (int i = 0; i < 15; i++)
        {
            int randomNumber = rand.Next(33, 126);
            temp[i] = (char)randomNumber;
        }
        s = new string(temp);
        textBox2.Text = s;
    }
  }
}
```

Εφαρμογή IsDebuggerPresent.cpp

Γλώσσα: Visual C#

Προγραμματιστής: Τζιώτζης Γαβριήλ

Σχόλια: Εισαγωγή κώδικα εντοπισμού εργαλείων αποσφαλμάτωσης στην Εφαρμογή SimpleValidate.cpp

Πηγαίος κώδικας:

```
#include <string>
#include <iostream>
#include <windows.h>
#include <stdio.h>

BOOL isDebuggerPresent();

#pragma comment(linker,"/FILEALIGN:512 /MERGE:.rdata=.text /MERGE:.data=.text /SECTION:.text,EWR /IGNORE:4078")
using namespace std;

static bool validate(string user, string password)
{
   if (user.length()<10)
      return false;
   else if (user.substr(2, 5).compare("admin")!=0)
   {
      cout<<user.substr(2,5)<<endl;
      return false;
   }
   else if (password.length()<15)
      return false;
```

```
    else
        return true;
};
static long factorial(int n)
{
    if (n==0)
        return 1;
    else if (n==1)
        return 1;
    else if (n>19)
    {
        cout<<"The number you entered is too big"<<endl;
        return -1;
    }
    else if (n<0)
    {
        cout<<"You can't count factorial for negative numbers"<<endl;
        return -2;
    }
    else
        return n*factorial(n-1);
};

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow )
{
    //Detecting Debugger
    if (isDebuggerPresent())
    {
```

```
    MessageBox(GetForegroundWindow(),"Debugger detected!\nReverse Engineering constitues a violation of the EULA.\nThe application will stop!","EULA Violation!",MB_ICONSTOP);
    return 0;
}
else
{
    MessageBox(GetForegroundWindow(),"No debugger detected!","OK!",MB_OK);
}

//Declaring the variables that will be used as containers for the information entered by the user.
string user;
string password;

cout<<"This progam will calculate factorial based on your inpu"<<endl;
cout<<"To use this program you must first login"<<endl<<endl;

//Requesting and reading username
cout<<"Please enter your username"<<endl;
cin>>user;

//Requesting and reading password
cout<<"Please enter your password"<<endl;
cin>>password;

//If username and password combination is not valid the system exits.
if (validate(user, password))
    cout<<"Welcome admin!"<<endl<<endl;
else
{
```

```
        cout<<"Invalid username or password. System will now terminate!"<<endl<<endl;
        return 1;
    }

    //Factorial calculation
    cout<<"Enter a negative number to exit"<<endl;
    long int n=-5;
    do
    {
        cout<<"Enter a number to calculate it's factorial"<<endl;
        cin>>n;
        if (!validate(user, password))
            return 2;
        if (n>=0)
            cout<<"Factorial of "<<n<<"= "<<factorial(n)<<endl<<endl;
        else
            cout<<"Thank you for using our program"<<endl;
    }
    while (n>=0);
};

BOOL isDebuggerPresent()
{
 BOOL result = FALSE;
 HINSTANCE kern_lib = LoadLibraryEx( "kernel32.dll", NULL, 0 );
        if( kern_lib ) {
                FARPROC lIsDebuggerPresent = GetProcAddress( kern_lib,
"IsDebuggerPresent" );
                if( lIsDebuggerPresent && lIsDebuggerPresent() ) {
                        result = TRUE;
                }
```

```
            FreeLibrary( kern_lib );
        }
        return result;
}
```