ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

«Ασφάλεια Ψηφιακών Συστημάτων»

Διπλωματική εργασία

"Προηγμένες τεχνικές αποφυγής των Antivirus"

Περίοδος 2016-2017

Επιμέλεια	***ΕΠΩΝΥΜΟ***	***ΟΝΟΜΑ***	***EMAIL***	***Α.Μ.***
	ΠΑΛΗΟΓΙΑΝΝΗΣ	ΠΑΝΑΓΙΩΤΗΣ	paliogiannis.panos@ssl-unipi.gr	MTE1528

Υπεύθυνος: Dr. Νταντογιάν Χριστόφορος

Περίληψη

Η τεχνική Return-Oriented Programming (ROP) θεωρείται ως μία πολυμορφική μέθοδος και μπορεί να χρησιμοποιηθεί ως μία εναλλακτική λύση για την αντικατάσταση των crypters και packers. Στα πλαίσια αυτής της διπλωματικής εργασίας, αξιοποιείται η τεχνική ROP για την αποφυγή των antivirus (AV) και πιο συγκεκριμένα, για την δημιουργία εκτελέσιμων αρχείων, τα οποία περιέχουν κάποιο γνωστό shellcode και παρόλα αυτά επιτυγχάνεται η επιτυχημένη διαφυγή τους από την σάρωση των antivirus. Με την τεχνική ROP, μπορεί να υπερνικηθεί η κύρια τεχνική σάρωσης που χρησιμοποιείται από τα antivirus και είναι ο μηχανισμός ανίχνευσης κακόβουλου κώδικα χρησιμοποιώντας υπογραφές (signatures) των ήδη ανιχνευμένων κακόβουλων προγραμμάτων. Στην παρούσα εργασία, αρχικά, παρατίθεται το θεωρητικό πλαίσιο το οποίο είναι απαραίτητο για την κατανόηση των τεχνικών θεμάτων τα οποία εξετάζονται στα κεφάλαια που ακολουθούν. Εν συνεχεία, αναλύονται οι τεχνικές που χρησιμοποιήθηκαν για την μετατροπή x86 κώδικα μηχανής (machine code) στην αντίστοιχη μορφή ROP καθώς και ο τρόπος δημιουργίας των εκτελέσιμων αρχείων που θα περιελάμβαναν τον συγκεκριμένο κώδικα. Επιπρόσθετα, παρουσιάζεται η αποτελεσματικότητα των τεχνικών αυτών όσον αφορά την συνέπεια με την οποία καταφέρνουν να αποφεύγουν τους μηχανισμούς ανίχνευσης των πιο γνωστών antivirus. Τέλος, ακολουθεί η ανάλυση του τρόπου λειτουργίας του προγράμματος που δημιουργήθηκε για την αυτοματοποιημένη μετατροπή ενός shellcode σε μορφή ROP καθώς επίσης, και οι βασικές διαδικασίες που επιτελεί για την παραγωγή ενός αξιόπιστου αποτελέσματος.

Πίνακας περιεχομένων

1. Εισαγωγή

Η ανάπτυξη των τεχνικών ανίχνευσης που χρησιμοποιούνται από τα antivirus έχει ως απαρχή την δημιουργία απλών υπογραφών (signatures), με κάθε υπογραφή να συνδέεται με έναν γνωστό ανιχνευμένο κακόβουλο κώδικα. Τα εκτελέσιμα αρχεία που αποθηκεύονται στον δίσκο, σαρώνονται πλέον από τα antivirus έτσι ώστε να ανιχνευθεί μία πιθανή ταυτοποίηση με τις υπογραφές που περιλαμβάνονται στην βάση δεδομένων τους. Ωστόσο οι συγγραφείς κακόβουλου κώδικα κατάφεραν σχετικά εύκολα να αποφύγουν τον συγκεκριμένο τρόπο ανίχνευσης, είτε εισάγοντας NOP (No-OPeration) εντολές είτε προσθέτοντας κώδικα ανάμεσα στις κακόβουλες εντολές, ο οποίος όμως δεν εκτελούταν ποτέ. Προκειμένου να αντιμετωπιστούν τέτοιου είδους δομές κακόβουλου κώδικα, οι απλές υπογραφές εξελίχθηκαν σε regular expressions (RegEx). Χρησιμοποιώντας regular expressions, τα antivirus είναι σε θέση να ανιχνεύουν κακόβουλο κώδικα ακόμα και στην περίπτωση που παρεμβάλλονται τυχαίες εντολές. Το επόμενο βήμα από την πλευρά των επιτιθέμενων, ήταν να κρυπτογραφούν τον κακόβουλο κώδικα (πολυμορφικός κώδικας) και να προχωράνε στην αποκρυπτογράφησή του αφού έχει αποκτηθεί ο πλήρης έλεγχος της ροής του στοχευμένου προγράμματος. Ωστόσο, για το τμήμα κώδικα (decryptor) που είναι υπεύθυνο για την αποκρυπτογράφηση όλου του υπόλοιπου λειτουργικού τμήματος του κακόβουλου κώδικα, μπορούν να δημιουργηθούν υπογραφές και επίσης, αυτός ο τύπος κακόβουλου λογισμικού, απαιτεί για την λειτουργία του ένα τμήμα στην μνήμη με πλήρη δικαιώματα (read, write and execute). Μία επαυξημένη περίπτωση του πολυμορφισμού είναι, ο μεταμορφικός κακόβουλος κώδικας, ο οποίος μπορεί να τροποποιεί την δομή του κάθε φορά που πρόκειται να εκτελεστεί, διατηρώντας όμως την ίδια λειτουργικότητα. Ο πολυμορφικός κώδικας μπορεί να ανιχνευθεί μελετώντας την εντροπία των εντολών μηχανής που περιέχει καθώς και με δυναμικούς τρόπους ανάλυσης, όπως για παράδειγμα, προσδιορισμός της συμπεριφοράς του σε ένα προστατευμένο και απομονωμένο περιβάλλον (sandbox). Μία αξιόπιστη εναλλακτική λύση σε πολυμορφικές μεθόδους, μπορεί να προσφέρει η τεχνική Return-Oriented Programming (ROP) η οποία αν συνδυαστεί με απλές τεχνικές για την αποφυγή της δυναμικής ανάλυσης που διεξάγουν τα antivirus, δύναται να παράγει αξιόπιστα αποτελέσματα.

2. Σχετικές εργασίες

Στην ενότητα αυτή, θα αναφέρουμε τα βασικά εργαλεία που έχουν αναπτυχθεί και στοχεύουν στην επίτευξη του ίδιου στόχου με αυτόν που έχει τεθεί στα πλαίσια της συγκεκριμένης εργασίας, δηλαδή την μόλυνση ενός PE αρχείου με κάποιο γνωστό shellcode και παράλληλα, την αποφυγή της ανίχνευσής του από τα υπάρχοντα antivirus.

Αρχικά, το εργαλείο Shellter [1], εστιάζει στην διατήρηση της αρχικής δομής του PE αρχείου, αποφεύγοντας την εισαγωγή του shellcode σε προκαθορισμένες διευθύνσεις του PE αρχείου ή την τροποποίηση των χαρακτηριστικών των ήδη υπαρχόντων τμημάτων του. Η διαδικασία που ακολουθεί είναι η επανεγγραφή κώδικα του PE αρχείου, για τον οποίο ισχύει η προϋπόθεση ότι θα του δοθεί ο έλεγχος κατά την διάρκεια εκτέλεσης του προγράμματος. Για να καθοριστεί εάν ισχύει η συγκεκριμένη προϋπόθεση διεξάγεται ανάλυση του εκτελέσιμου αρχείου καθώς και της ροής εκτέλεσής του. Επίσης, το εργαλείο Shellter είναι σε θέση να αλλάζει τα δικαιώματα του τμήματος μνήμης στο οποίο εισάγεται το shellcode. Με αυτόν τον τρόπο μπορεί να χρησιμοποιηθεί ένα κρυπτογραφημένο shellcode ή κώδικας που αλλάζει δυναμικά την δομή του. Ακόμα, έχει την δυνατότητα να εισάγει περιττό κώδικα έτσι ώστε να καθυστερεί η εκτέλεση του επιθυμητού shellcode και επομένως να αποφεύγεται τυχόν ανίχνευσή του, από τις δυναμικές μεθόδους των antivirus. Το συγκεκριμένο εργαλείο, χρησιμοποιεί πολύπλοκες τεχνικές προκειμένου να εντοπίσει το κατάλληλο σημείο της μνήμης στο οποίο θα εισαχθεί το shellcode. Παρόλα αυτά, μπορεί να αναφερθεί ως μειονέκτημα το γεγονός ότι βασίζεται σε πολυμορφικές μεθόδους και επομένως παραμένει ευάλωτο, όσον αφορά την ανίχνευσή του από τις υπογραφές που χρησιμοποιούν τα antivirus. Επιπρόσθετα, όπως αναφέρθηκε προηγουμένως, σε πολλές περιπτώσεις τροποποιεί τα δικαιώματα μνήμης του τμήματος .text, και ως εκ τούτου θα μπορούσε να εντοπιστεί εξαιτίας τέτοιου είδους αλλαγών.

Το PEinject [2] είναι πιο ορθό να χαρακτηριστεί ως μία μέθοδος παρά ως ένα ολοκληρωμένο εργαλείο. Υπολογίζει τον διαθέσιμο χώρο μνήμης στο τμήμα .text και αφού βρεθεί μία περιοχή που καλύπτει τις ανάγκες του shellcode, το εισάγει στο συγκεκριμένο τμήμα της μνήμης. Δεν υποστηρίζει δυνατότητες κρυπτογράφησης ή τροποποίησης του shellcode καθ' οποιονδήποτε τρόπο, προτού προχωρήσει στην εισαγωγή του shellcode στην μνήμη του εκτελέσιμου αρχείου. Προκειμένου να δοθεί ο έλεγχος στο εισαγόμενο shellcode, τροποποιεί την διεύθυνση του κατάλληλου πεδίου (entry point) της NT_HEADER του εκτελέσιμου αρχείου.

Το εργαλείο ROPInjector [3] μετατρέπει ένα shellcode στο ROP ισοδύναμό του και το εισάγει σε ένα δεδομένο PE αρχείο. Η ROP αλυσίδα που δημιουργείται δανείζεται εντολές που περιέχονται ήδη στον κώδικα του PE αρχείου και έτσι, αποφεύγεται η δημιουργία μοτίβων καθώς και ο επηρεασμός της εντροπίας του αρχικού PE αρχείου. Επιπρόσθετα, ο κώδικας ROP που παράγεται, είναι διαφορετικός δεδομένων διαφορετικών PE αρχείων και επομένως είναι δύσκολη η διαδικασία παραγωγής υπογραφών από την πλευρά των antivirus.

Τέλος, έχουμε το σύστημα παραγωγής καμουφλαρισμένων εκτελέσιμων αρχείων, το οποίο ονομάζεται Frankenstein [4]. Το συγκεκριμένο σύστημα απέχει από την λογική των μεταμορφικών μηχανών και για την παραγωγή των εκτελέσιμων αρχείων που προσφέρουν την επιθυμητή λειτουργικότητα, δημιουργεί μία δομή που αποτελείται από εντολές οι οποίες έχουν εντοπιστεί σε υπάρχοντα προγράμματα του στοχευμένου συστήματος. Το σημαντικότερο πλεονέκτημα της συγκεκριμένης τεχνικής είναι ότι τα προγράμματα αυτά κατηγοριοποιούνται ως μη κακόβουλα από τους τοπικούς αμυντικούς μηχανισμούς του στοχευμένου συστήματος. Έτσι, καθίσταται πιο δύσκολη για τα antivirus η ανίχνευση του κακόβουλου κώδικα, βάσει συγκεκριμένων ακολουθιών από bytes (signature-based detection). Η διαδικασία που χρησιμοποιεί το σύστημα Frankenstein για την εύρεση των απαραίτητων εντολών, εκμεταλλεύεται τις τελευταίες ανακαλύψεις που συντελέστηκαν στην τεχνική ROP και ειδικότερα, στον τρόπο εύρεσης των gadgets. Τα πειράματα των ερευνητών επικεντρώθηκαν κυρίως στην συγκάλυψη κώδικα του οποίου το μήκος ήταν σχετικά μικρό, όπως για παράδειγμα οι unpackers. Αντίθετα, με τον συνηθισμένο ορισμό ενός gadget, δηλαδή μία ακολουθία εντολών που τελειώνει με μία εντολή επιστροφής ret, στα πλαίσια της συγκεκριμένης ερευνητικής εργασίας, ως gadget ορίσθηκε κάθε έγκυρη ακολουθία x86 εντολών καθώς η ένωση των gadget γίνεται με στατικό τρόπο, οπότε δεν υπάρχει κάποια υποχρέωση τήρησης του συγκεκριμένου περιορισμού. Κατά την διαδικασία ανίχνευσης των gadgets, το σύστημα Frankenstein σαρώνει τα τοπικά δυαδικά αρχεία του στοχευμένου συστήματος για την εύρεση όλων των χρήσιμων εντολών που ενδέχεται να περιέχουν. Σύμφωνα με τα αποτελέσματα των ερευνητών, επαρκεί η σάρωση 2-3 δυαδικών αρχείων του system32 φακέλου για να δημιουργηθεί ένα αρκετά μεγάλο σύνολο από gadgets που μπορούν να παρέχουν Turing-complete λειτουργικότητα. Το σύστημα Frankenstein συλλέγει byte ακολουθίες από τα δυαδικά αρχεία χρησιμοποιώντας ένα μεταβλητό παράθυρο και στην συνέχεια, οι ακολουθίες αυτές αποκωδικοποιούνται για την παραγωγή των αντίστοιχων εντολών. Για την επίτευξη της επιθυμητής λειτουργικότητας, χρησιμοποιούνται τα κατάλληλα gadgets τα οποία ενσωματώνονται εν συνεχεία, σε μία προϋπάρχουσα δομή δυαδικού αρχείου. Με αυτόν τον τρόπο δημιουργούνται ισοδύναμης λειτουργικότητας εκτελέσιμα αρχεία, τα οποία όμως αποτελούνται από ένα διαφορετικό σύνολο εντολών το καθένα. Η κύρια συνεισφορά της συγκεκριμένης εργασίας είναι η εισαγωγή της ιδέας ότι η δημιουργία εκτελέσιμων αρχείων τα οποία αποτελούνται από ακολουθίες εντολών που έχουν συλλεχθεί από μη κακόβουλα προγράμματα, μπορεί να αυξήσει σημαντικά την δυσκολία εντοπισμού του κακόβουλου κώδικα από τους υπάρχοντες αμυντικούς μηχανισμούς.

3. Βασικό θεωρητικό πλαίσιο

Στο παρών κεφάλαιο, θα αναφερθούμε στις βασικές θεωρητικές έννοιες καθώς και σε λίγο πιο πρακτικά ζητήματα, τα οποία όμως είναι απαραίτητα για την κατανόηση των κεφαλαίων που ακολουθούν. Πιο συγκεκριμένα, θα αναλυθεί η έννοια του Return-Oriented Programming (ROP) και εν συνεχεία, θα αναφερθούμε στην συνηθέστερη ευπάθεια των σύγχρονων προγραμμάτων, δηλαδή στην υπερχείλιση της στοίβας (buffer overflow). Τέλος, θα αναλύσουμε τον τρόπο λειτουργίας ενός shellcode καθώς και το πώς αλληλεπιδρά ένα shellcode με το λειτουργικό σύστημα και κατ' επέκταση πως είναι θέση να καλεί χρήσιμες συναρτήσεις έτσι ώστε να εξυπηρετήσει την λειτουργία του.

3.1 Return-Oriented Programming

Η τεχνική Return-Oriented Programming δημοσιεύθηκε το 2007 από τον Hovav Shacham [5] ως μία προχωρημένη μέθοδος για την εκμετάλλευση ευπαθειών που εντοπίζονται στην στοίβα (stack) ενός προγράμματος και μπορούν να συντελέσουν στον έλεγχο της ροής του προγράμματος από τον εκάστοτε επιτιθέμενο. Η κύρια επιδίωξη της τεχνικής αυτής ήταν να υπερκεράσει τον αποτελεσματικότερο μηχανισμό άμυνας εναντίων τέτοιων επιθέσεων, ο οποίος είναι γνωστός με την ονομασία Data Execution Prevention (DEP), και να δώσει στον επιτιθέμενο την δυνατότητα να εκτελέσει οποιουσδήποτε υπολογισμούς επιθυμούσε.

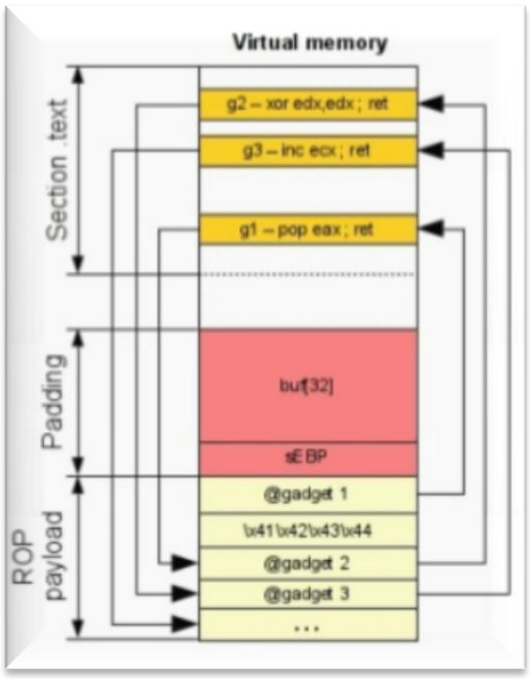
Ας θεωρήσουμε έναν επιτιθέμενο, ο οποίος έχει ανακαλύψει μία ευπάθεια σε κάποιο πρόγραμμα και επιθυμεί να την εκμεταλλευτεί. Στο πλαίσιο που μελετάμε, με τον όρο εκμετάλλευση (exploitation), εννοούμε τον έλεγχο της ροής του συγκεκριμένου προγράμματος από τον επιτιθέμενο, έτσι ώστε να είναι σε θέση να πραγματοποιήσει τις ενέργειες της επιλογής του, ανεξάρτητα από το επίπεδο σημαντικότητας των διαπιστευτηρίων (credentials) που κατέχει. Η πιο συνηθισμένη ευπάθεια είναι γνωστή ως buffer overflow στην στοίβα του προγράμματος, χωρίς να αποτελεί την μοναδική, καθώς υπάρχουν αρκετές γνωστές ευπάθειες, από τις οποίες ξεχωρίζουν τα buffer overflows στον σωρό (heap) του προγράμματος και τα integer overflows. Σε κάθε περίπτωση, ο επιτιθέμενος πρέπει να φέρει σε πέρας δύο βασικές εργασίες:

- Την ανακατεύθυνση της ροής του προγράμματος και συνεπακόλουθα τον πλήρη έλεγχό της.
- Την εκτέλεση των υπολογισμών που επιθυμεί, επιτυγχάνοντας την χειραγώγηση της συμπεριφοράς του στοχευμένου προγράμματος.

Στις περισσότερες stack-smashing επιθέσεις, ο κακόβουλος χρήστης επιτυγχάνει την πρώτη βασική εργασία, επανεγγράφοντας την διεύθυνση επιστροφής (return address) στην στοίβα του προγράμματος, έτσι ώστε να ανακατευθύνει την κανονική ροή εκτέλεσης σε κώδικα της επιλογής του. Για την επίτευξη του δεύτερου βήματος, θα πρέπει να εισάγει τον κώδικα που θέλει εν τέλει να εκτελεστεί, μετά την ανακατεύθυνση της ροής του προγράμματος. Εξαιτίας της συμπεριφοράς των ρουτινών της γλώσσας προγραμματισμού C, που είναι υπεύθυνες για τον χειρισμό των αλφαριθμητικών (strings) και από τις οποίες προέρχονται οι ευπάθειες, ο εισαχθέν κώδικας δεν θα πρέπει να περιέχει NUL bytes. Η τεχνική ROP δίνει στον επιτιθέμενο τα κατάλληλα εργαλεία για την υπερνίκηση των μέτρων ασφάλειας που

στοχεύουν στο να δυσκολέψουν σε μεγάλο βαθμό την επιτυχημένη περάτωση του δεύτερου βήματος.

Η τεχνική ROP παρέχει μία πλήρως λειτουργική "γλώσσα" στον επιτιθέμενο, με την οποία είναι σε θέση να προσομοιώσει τις βασικότερες προγραμματιστικές δομές ενάντια σε εφαρμογές-στόχους που συνδέονται με την βασική βιβλιοθήκη της C και με την προϋπόθεση ότι περιέχουν μία ευπάθεια τύπου buffer overflow, την οποία δύναται να εκμεταλλευτεί ο επιτιθέμενος με κάποιον τρόπο. Στην συνέχεια, δημιουργεί μία αλυσίδα από διευθύνσεις στην στοίβα του προγράμματος, που δείχνουν σε συγκεκριμένα τμήματα εντολών του στοχευμένου εκτελέσιμου ή των βιβλιοθηκών που έχουν φορτωθεί. Κάθε τέτοιο τμήμα θα πρέπει να τελειώνει με μία εντολή επιστροφής, με την πιο συνηθισμένη να είναι η ret, έτσι ώστε να διατηρείται ο έλεγχος της ροής του προγράμματος. Αυτά τα μικρά τμήματα εντολών είναι γνωστά ως gadgets και η εντολή επιστροφής που περιέχουν, είναι στην πραγματικότητα μία κλήση στο επόμενο gadget που έχει προστεθεί στην υπάρχουσα αλυσίδα. Ως ένας παραλληλισμός με την εκτέλεση τυπικού κώδικα, στην τεχνική ROP, τα gadgets είναι οι εντολές και ο καταχωρητής esp έχει τον ρόλο της διευθέτησης της σειράς των εντολών που πρόκειται να εκτελεστούν. Στο παρακάτω στιγμιότυπο απεικονίζεται η εκτέλεση μιας αλυσίδας αλληλένδετων gadgets. Η αλυσίδα αποτελείται από διευθύνσεις που δείχνουν σε συγκεκριμένες θέσεις μνήμης, στις οποίες εντοπίζονται οι επιθυμητές εντολές. Η θέση των gadgets στην δημιουργηθείσα αλυσίδα, δίνει και την σειρά εκτέλεσής τους.



Εικόνα 1: Παράδειγμα εκτέλεσης return-oriented programming εντολών.

Όπως έχουμε αναφέρει, τα βασικά δομικά στοιχεία είναι μικρά τμήματα κώδικα που αποτελούνται συνήθως από δύο ή τρεις εντολές. Το κύριο πλεονέκτημα της τεχνικής ROP βασίζεται στην ίδια την αρχιτεκτονική των συστημάτων και πιο συγκεκριμένα στον τρόπο με τον οποίο μπορούν να ερμηνευτούν οι εντολές assembly σε x86 εκτελέσιμα. Πολλές από τις ακολουθίες εντολών που μπορούν να χρησιμοποιηθούν από τον επιτιθέμενο,

δημιουργούνται από τις επιλογές του μεταγλωττιστή (compiler) κατά την παραγωγή του κώδικα. Όμως, εξίσου σημαντικές είναι και εντολές που μπορούν να εντοπιστούν στην βιβλιοθήκη libc, οι οποίες στην πραγματικότητα δεν έχουν τοποθετηθεί εκεί από τον μεταγλωττιστή. Σε κάθε περίπτωση, αυτές οι ακολουθίες εντολών είναι εξαιρετικά δύσκολο να εξαλειφθούν χωρίς να γίνουν εκτεταμένες τροποποιήσεις τόσο στον μεταγλωττιστή όσο και στον assembler.

Η x86 αρχιτεκτονική παρουσιάζει ένα ευρύ σύνολο διαθέσιμων εντολών, γεγονός που μπορεί να εξηγήσει πως μπορούν να βρεθούν από τον επιτιθέμενο, εντολές που δεν έχουν δημιουργηθεί από τον ίδιο τον μεταγλωττιστή. Επίσης, το σύνολο εντολών της αρχιτεκτονικής x86 είναι εξαιρετικά πυκνό και ως εκ τούτου, μία τυχαία ακολουθία από bytes μπορεί να ερμηνευτεί ως έγκυρες εντολές με υψηλή πιθανότητα. Τέλος, για να είναι μία ακολουθία χρήσιμη σε μία πιθανή επίθεση, η μόνη προϋπόθεση είναι να λήγει σε μία διεύθυνση επιστροφής, η οποία αναπαρίσταται από το byte “c3”. Για την καλύτερη κατανόηση, ας υποθέσουμε τις εξής δύο εντολές:

f7 c7 07 00 00 00	test $0x00000007, %edi
0f 95 45 c3	setnzb – 61(%ebp)

Αν ο επιτιθέμενος ξεκινήσει να ερμηνεύει τις εντολές παραλείποντας το πρώτο byte, τότε το αποτέλεσμα είναι το παρακάτω:

c7 07 00 00 00 0f	movl $0x0f000000, (%edi)
95	xchg %ebp, %eax
45	inc %ebp
c3	ret

Δηλαδή, τα ίδια bytes στην μνήμη μπορούν να αντιστοιχηθούν σε διαφορετικές εντολές assembly. Δεν τίθεται κάποιος περιορισμός από την αρχιτεκτονική σχεδίαση όσον αφορά το πρώτο byte που θα πρέπει να διαβαστεί. Η συχνότητα με την οποία εμφανίζονται τέτοιου είδους εντολές εξαρτάται από τα χαρακτηριστικά της γλώσσας που εξετάζεται.

3.2 Καταχωρητές του x86 επεξεργαστή

Ο πρώτος x86 επεξεργαστής ήταν η 8086 CPU. Σχεδιάστηκε και αναπτύχθηκε από την Intel, η οποία αργότερα προχώρησε στην παραγωγή πιο προχωρημένων επεξεργαστών της ίδιας οικογένειας [6]. Ο x86 επεξεργαστής διαθέτει αρκετούς καταχωρητές (registers) οι οποίοι είναι σαν εσωτερικές μεταβλητές για τον επεξεργαστή.

Οι καταχωρητές EAX, ECX, EDX και EBX είναι γνωστοί ως γενικού σκοπού καταχωρητές και φέρουν τις ονομασίες Accumulator, Counter, Data και Base καταχωρητής, αντίστοιχα. Εξυπηρετήσουν διάφορους σκοπούς, αλλά κυρίως χρησιμοποιούνται από την CPU ως προσωρινές μεταβλητές κατά την εκτέλεση εντολών μηχανής.

Οι καταχωρητές ESP, EBP, ESI και EDI είναι γνωστοί, επίσης, ως γενικού σκοπού καταχωρητές, αλλά μερικές φορές αναφέρονται και ως δείκτες (pointers/indexes). Φέρουν τις ονομασίες Stack Pointer, Base Pointer, Source Index και Destination Index, αντίστοιχα. Οι πρώτοι δύο καταχωρητές καλούνται δείκτες επειδή αποθηκεύουν διευθύνσεις μήκους 32-bit και είναι υπεύθυνοι για την διατήρηση συγκεκριμένων θέσεων μνήμης της στοίβας του

προγράμματος. Ο ρόλος που επιτελούν είναι ιδιαίτερα σημαντικός για την σωστή εκτέλεση του προγράμματος και την αποτελεσματική διαχείριση της μνήμης. Οι τελευταίοι δύο καταχωρητές χρησιμοποιούνται συνήθως ως δείκτες πηγής και προορισμού όταν απαιτείται να διαβαστούν ή να εγγραφούν δεδομένα.

Ο EIP καταχωρητής καλείται Instruction Pointer και δείχνει την τρέχουσα εντολή που πρόκειται να διαβαστεί από τον επεξεργαστή και στην συνέχεια να εκτελεστεί. Ο επεξεργαστής διαβάζει κάθε εντολή χρησιμοποιώντας τον καταχωρητή EIP ως δείκτη. Τέλος, ο EFLAGS καταχωρητής αποτελείται στην πραγματικότητα από αρκετά bit τα οποία λειτουργούν ως σημαίες (flags) και χρησιμοποιούνται κυρίως για την διεξαγωγή συγκρίσεων.

3.3 Stack-based buffer overflow

Ιστορικά, η εκμετάλλευση ευπαθειών τύπου buffer overflow, υπήρξε η πιο διαδεδομένη μέθοδος για το “χακάρισμα” λογισμικού. Η γνώση τέτοιων ευπαθειών, υπάρχει θεωρητικά, όσο και η ίδια η γλώσσα προγραμματισμού C, ενώ τέτοιου είδους επιθέσεις ανιχνεύονται για πάνω από 25 χρόνια. Παρά το γεγονός ότι, η buffer overflow ευπάθεια είναι πλήρως κατανοητή και έχει τεκμηριωθεί από πάμπολλες δημοσιεύσεις, παραμένει ως η πιο συνηθισμένη αδυναμία των σύγχρονων εφαρμογών από την οποία μπορεί να επωφεληθεί ένας κακόβουλος χρήστης.

Η γλώσσα προγραμματισμού C είναι μία γλώσσα υψηλού επιπέδου, η οποία για να λειτουργήσει αξιόπιστα, προϋποθέτει ότι ο προγραμματιστής είναι υπεύθυνος για τον έλεγχο της εγκυρότητας των δεδομένων. Στην περίπτωση που αυτή η εργασία μετατοπιζόταν στην πλευρά του μεταγλωττιστή, τότε τα παραγόμενα δυαδικά αρχεία θα είχαν σημαντικά πιο αργούς χρόνους εκτέλεσης, εξαιτίας των ελέγχων που θα έπρεπε να διεξάγονται για κάθε μεταβλητή του προγράμματος. Ακόμα, θα είχε σαν αποτέλεσμα την μείωση του επιπέδου ελέγχου που έχει ο προγραμματιστής και την περαιτέρω αύξηση της πολυπλοκότητας της ίδιας της γλώσσας.

Παρόλο που η απλότητα της γλώσσας προγραμματισμού C αυξάνει τον έλεγχο που έχει ο προγραμματιστής και συντελεί στην παραγωγή αποτελεσματικότερων προγραμμάτων, μπορεί επίσης να έχει ως αποτέλεσμα προγράμματα τα οποία είναι ευπαθή σε buffer overflows, στην περίπτωση που ο προγραμματιστής δεν είναι αρκετά προσεκτικός. Αυτό σημαίνει ότι, από την στιγμή που θα διανεμηθεί μνήμη σε μία μεταβλητή, δεν υπάρχουν πρόσθετοι εγγενείς μηχανισμοί ασφάλειας για την διασφάλιση ότι τα δεδομένα που πρόκειται να αποθηκευτούν, χωράνε στην δεσμευμένη μνήμη. Αν για παράδειγμα, ο προγραμματιστής προσπαθήσει να αποθηκεύσει δεδομένα μήκους δέκα bytes σε έναν buffer για τον οποίο έχουν δεσμευτεί μόνο οχτώ bytes μνήμης, η ενέργεια αυτή θα ολοκληρωθεί με επιτυχία, ακόμα και αν έχει ως αποτέλεσμα τον πρόωρο τερματισμό του προγράμματος λόγω μοιραίου σφάλματος. Τέτοιου είδους αστοχίες ονομάζονται buffer overruns ή buffer overflows, από την στιγμή που τα δύο επιπλέον bytes δεδομένων πρόκειται να υπερχειλίσουν την διαθέσιμη κατανεμημένη μνήμη, επανεγγράφοντας οτιδήποτε ακολουθεί. Αν τα δεδομένα που χαθούν λόγω της υπερχείλισης είναι κρίσιμα για την σωστή λειτουργία του προγράμματος, τότε το πρόγραμμα θα τερματιστεί πάραυτα.

Ένας επιτιθέμενος μπορεί να εκμεταλλευτεί μία πιθανή αδυναμία τύπου buffer overflow, έτσι ώστε να αποκτήσει τον έλεγχο ροής εκτέλεσης του προγράμματος, δηλαδή να καταφέρει να αποκτήσει τον έλεγχο του καταχωρητή EIP, επανεγγράφοντας μία διεύθυνση επιστροφής στην στοίβα του προγράμματος. Με αυτόν τον τρόπο μπορεί να εκτελέσει δικό του κώδικα (shellcode), ανακατευθύνοντας την ροή του προγράμματος στην διεύθυνση μνήμης που έχει αποθηκευτεί ο επιθυμητός προς εκτέλεση κώδικας.

3.4 Βασικοί μηχανισμοί προστασίας

Μετά την ραγδαία ανάπτυξη των τεχνικών εκμετάλλευσης αδυναμιών που έφεραν τα διάφορα προγράμματα, οι μεγάλες εταιρίες που ήταν υπεύθυνες για την δημιουργία των λειτουργικών συστημάτων, άρχισαν να προσθέτουν μηχανισμούς προστασίας έτσι ώστε να προστατέψουν τους απλούς χρήστες. Όλοι αυτοί οι μηχανισμοί προστασίας έχουν ως στόχο την μείωση της πιθανότητας επιτυχίας μίας πιθανής επίθεσης, χωρίς όμως να εξαφανίζουν την υπάρχουσα ευπάθεια. Στις υποενότητες που ακολουθούν, περιγράφουμε τους βασικούς μηχανισμούς προστασίας των σύγχρονων λειτουργικών συστημάτων και σε κάποιες περιπτώσεις αναφέρονται πιθανοί τρόποι ώστε να καταστούν αναποτελεσματικοί σε μία πιθανή επίθεση.

3.4.1 Non-Executable Stack

Ένα πολύ συνηθισμένο κενό ασφαλείας των προγραμμάτων, όπως ήδη αναφέρθηκε, είναι τα stack-based buffer overflow, με τον πιο διαδεδομένο τρόπο εκμετάλλευσης αυτής της ευπάθειας να είναι η τοποθέτηση κώδικα στην στοίβα του προγράμματος. Ο κώδικας αποθηκεύεται στον ίδιο buffer, στον οποίο έγινε η υπερχείλιση, επανεγγράφοντας την διεύθυνση επιστροφής και εν συνεχεία, καλώντας τον αποθηκευμένο κώδικα. Προκειμένου να αντιμετωπιστεί αυτή η τεχνική επίθεσης, δημιουργήθηκε το αντίμετρο με την ονομασία non-executable stack.

Οι περισσότερες εφαρμογές δεν εκτελούν ποτέ κώδικα στην στοίβα του προγράμματος, οπότε το συγκεκριμένο μέτρο ασφάλειας υπαγορεύει ότι δεν πρέπει να είναι δυνατή η εκτέλεση οποιασδήποτε εντολής στην στοίβα. Με την ενεργοποίηση του non-executable stack, καθίσταται αναποτελεσματική η εισαγωγή του shellcode σε οποιοδήποτε σημείο της στοίβας του προγράμματος.

Τα σύγχρονα λειτουργικά συστήματα όπως είναι: διανομές Linux, Windows, OpenBSD, Mac OS X και Solaris, ενσωματώνουν το συγκεκριμένο χαρακτηριστικό στην λίστα των μέτρων ασφάλειας που υλοποιούν και προσφέρουν.

Με την έλευση του non-executable stack αντίμετρου, δημιουργήθηκαν αρκετές νέες τεχνικές επίθεσης για την παράκαμψή του. Όλες αυτές οι τεχνικές βασίστηκαν στο γεγονός ότι μόνο η στοίβα του προγράμματος χαρακτηρίζεται ως “μη-εκτελέσιμη”. Οπότε θα μπορούσε να εκτελεστεί κώδικας ο οποίος υπάρχει σε ένα άλλο σημείο του εκτελέσιμου πέραν της στοίβας. Το βασικό βήμα παρέμενε η επανεγγραφή της διεύθυνσης επιστροφής με την κατάλληλη τιμή. Η πρώτη τεχνική επίθεσης που καθιστούσε την μη-εκτελέσιμη στοίβα αναποτελεσματική, φέρει την ονομασία return-into-libc (ret2libc), με την τεχνική ROP να ακολουθεί μερικά χρόνια αργότερα.

3.4.2 W ^ X (Εγγράψιμη ή εκτελέσιμη) μνήμη

Αποτελεί μία λογική επέκταση της μη εκτελέσιμης στοίβας και υπαγορεύει ότι περιοχές της μνήμης που χαρακτηρίζονται ως εγγράψιμες δεν μπορούν ταυτόχρονα να επιτρέπουν την εκτέλεση κώδικα σε αυτό το σημείο της μνήμης. Αντίστοιχα, σε περιοχές που επιτρέπεται η εκτέλεση κώδικα, θα πρέπει να μην είναι δυνατή η εγγραφή οποιουδήποτε είδους δεδομένων. Θεωρητικά, με την ενεργοποίηση του συγκεκριμένου μηχανισμού προστασίας, είναι αδύνατη η εισαγωγή κώδικα σε ένα στοχευμένο ευπαθές πρόγραμμα. Ωστόσο, κάτι τέτοιο δεν ισχύει σε πραγματικά περιβάλλοντα.

Μπορεί να βρεθεί ένας μεγάλος αριθμός διαφορετικών υλοποιήσεων του W ^ X μηχανισμού προστασίας, παρόλα αυτά η πρώτη υλοποίηση που φαίνεται να ξεχωρίζει εντοπίζεται ως μία τροποποίηση του λειτουργικού συστήματος Linux και φέρει την ονομασία PaX. Η συγκεκριμένη υλοποίηση έγινε γνωστή τον Οκτώβριο του έτους 2000. Ήταν η πρώτη υλοποίηση που απέδειξε, ότι αντίθετα από την κοινή αποδεκτή πεποίθηση, είναι δυνατή η επίτευξη των μη εκτελέσιμων σελίδων μνήμης σε Intel x86 υλικό (hardware). Παρά το γεγονός ότι η υλοποίηση αυτή λειτούργησε σωστά, αντικαταστάθηκε αργότερα από μία πιο συντηρητική λύση και δεν συμπεριλήφθηκε σε καμία κύρια έκδοση Linux, κυρίως για λόγους απόδοσης και όχι για λόγους που αφορούν το παρεχόμενο επίπεδο ασφάλειας.

Τον Σεπτέμβριο του 2003, η AMD εισήγαγε σε τσιπ της, υποστήριξη για μη εκτελέσιμες σελίδες μνήμες, με το χαρακτηριστικό αυτό να ονομάζεται “NX” (Non-eXecutable). Αμέσως μετά, ακολούθησε η Intel με ένα παρόμοιο χαρακτηριστικό ασφαλείας το οποίο καλούταν “ED” (Execute Disable). Μερικούς μήνες αργότερα το χαρακτηριστικό αυτό υλοποιείται σε Linux διανομές καθώς επίσης διατίθεται για τα Windows XP με το Service Pack 2. Η Microsoft ονομάζει το συγκεκριμένο αντίμετρο ως Data Execution Prevention (DEP).

Όταν ο W ^ X μηχανισμός προστασίας υλοποιείται κατάλληλα, είναι εξαιρετικά αποτελεσματικός στην αποτροπή επιθέσεων που στοχεύουν στην εκτέλεση κώδικα που δεν συμπεριλαμβάνεται στο αρχικό πρόγραμμα. Ωστόσο, αυτό δεν σημαίνει ότι δεν μπορεί να εκτελεστεί κώδικας καθ’ οποιονδήποτε τρόπο. Στην περίπτωση του stack-based buffer overflow, η χρήση της τεχνικής ret2code επαρκεί για την εκτέλεση των περισσότερων επιθέσεων.

3.4.3 Canaries

Τα canaries ή cookies όπως αλλιώς ονομάζονται, είναι συνήθως τιμές των 32-bit και τοποθετούνται ανάμεσα στους buffers και τις ευαίσθητες πληροφορίες. Στην περίπτωση που συμβεί κάποια υπερχείλιση ενός buffer, η τιμή του canary επανεγγράφεται και έτσι, μπορεί να εντοπιστεί από την εφαρμογή η πιθανή επίθεση.

Η αρχική ιδέα προέρχεται από τον Crispin Cowan και παρουσιάστηκε τον Δεκέμβριο του 1997 με την ονομασία StackGuard. Ακολούθησε τον Ιούνιο του 2000 ο Hiroaki Etoh, ο οποίος βελτίωσε σημαντικά την αρχική ιδέα εισάγοντας παράλληλα την ονομασία Stack-Smashing Protector (SSP).

Όταν παρουσιάστηκε η ιδέα του StackGuard, οι περισσότερες stack-based buffer overflow επιθέσεις πραγματοποιούνταν επανεγγράφοντας την διεύθυνση επιστροφής. Ως φυσική απόρροια του γεγονότος αυτού, η αρχική ιδέα ήταν να προστατευτεί μόνο η διεύθυνση επιστροφής στην στοίβα του προγράμματος. Ο Tim Newsham απέδειξε πολύ γρήγορα ότι το StackGuard είναι ένα αναποτελεσματικό αντίμετρο όταν άλλες τοπικές μεταβλητές περιέχουν ευαίσθητες πληροφορίες. Ακολούθησε ο Gerardo Richarte, το έτος 2002, ο οποίος παρουσίασε τρόπους για να υπερνικηθεί ο μηχανισμός του StackGuard στις περισσότερες των περιπτώσεων. Οι ευπάθειες του StackGuard δεν διορθώθηκαν ποτέ και πλέον δεν χρησιμοποιείται. Ως αντικαταστάτες του θεωρούνται ο μηχανισμός Stack-Smashing Protector όσον αφορά τον GCC μεταγλωττιστή και ο μηχανισμός προστασίας /GS για το Microsoft Visual Studio.

Το πρώτο canary που χρησιμοποιήθηκε είναι γνωστό ως NUL canary, και αποτελούταν από μηδενικά (0x00000000) ενώ ακολούθησε η αντικατάστασή του από το terminator canary (0x000aff0d). Το τελευταίο είδος canary, περιελάμβανε την τιμή '\x00' έτσι ώστε να σταματάει συναρτήσεις όπως η strcpy(), τις τιμές '\x0d' και '\x0a' για να σταματάει συναρτήσεις όπως η gets() καθώς και την τιμή '\xff' (EOF) για την αποτροπή κάποιων άλλων συναρτήσεων από την υπερχείλιση του buffer και ακολούθως, την επανεγγραφή της διεύθυνσης επιστροφής. Όσον αφορά το terminator canary, αν ο επιτιθέμενος προσπαθήσει να επανεγγράψει στην θέση μνήμης που εντοπίζεται το canary, την αρχική τιμή του, η εκάστοτε συνάρτηση θα σταματήσει να γράφει δεδομένα στον buffer και έτσι θα αποτραπεί η υπερχείλισή του. Τέλος, υπάρχει και το random canary το οποίο όπως υπαγορεύει και το όνομά του, αποτελείται από μία τυχαία τιμή. Για να το υπερνικήσει κάποιος επιτιθέμενος θα πρέπει είτε να διαβάσει την τιμή του με κάποιον τρόπο από την μνήμη είτε να καταφέρει να προβλέψει την τιμή αυτή.

3.4.4 Address Space Layout Randomization (ASLR)

Η αρχή πίσω από τον μηχανισμό προστασίας ASLR είναι απλή: εάν όλες οι πιθανές διευθύνσεις δηλαδή των βιβλιοθηκών, της ίδιας της εφαρμογής, της στοίβας καθώς και του σωρού, είναι τυχαίες, τότε ο επιτιθέμενος δεν θα είναι σε θέση να γνωρίζει εκ των προτέρων την τοποθεσία στην οποία θα πρέπει να μεταβεί ώστε να εκτελέσει τον κώδικα της επιλογής του.

Ωστόσο, ο ASLR μηχανισμός, δεν αποτελεί ένα αντίμετρο το οποίο μπορεί να λειτουργήσει αποτελεσματικά ενάντια σε κάθε πιθανό είδος επίθεσης. Στην περίπτωση που ο επιτιθέμενος μπορεί να εισάγει τον κώδικα της αρεσκείας του στην εκτελέσιμη μνήμη του προγράμματος και υπάρχει αρκετός υπολειπόμενος χώρος μνήμης για την εισαγωγή ενός σημαντικού αριθμού από εντολές NOPS, τότε ενδέχεται να καταφέρει την εκτέλεση του κώδικα που εισήγαγε. Επιπρόσθετα, η αρχιτεκτονική των ίδιων των συστημάτων, δεν επιτρέπει την πλήρη τυχαιοποίηση των διευθύνσεων και έτσι, ο επιτιθέμενος έχει αυξημένες πιθανότητες να αποφύγει τον συγκεκριμένο μηχανισμό ασφαλείας, χρησιμοποιώντας την επίθεση της εξαντλητικής αναζήτησης (brute force attack).

3.5 Shellcode

Ως shellcode ορίζεται ένα σετ εντολών, το οποίο εισάγεται στην μνήμη ενός ευάλωτου προγράμματος και στην συνέχεια εκτελείται από αυτό. Το εκάστοτε shellcode χρησιμοποιείται για την απευθείας τροποποίηση της τιμής των καταχωρητών και την χειραγώγηση της συνολικής λειτουργίας του προγράμματος [7].

Γενικά, για την συγγραφή ενός shellcode απαιτείται η χρήση ενός assembler και στην συνέχεια, η εξαγωγή των αντίστοιχων δεκαεξαδικών opcodes. Στις περισσότερες των περιπτώσεων, το shellcode θα πρέπει να παραχθεί χρησιμοποιώντας την γλώσσα Assembly και όχι κάποια γλώσσα προγραμματισμού υψηλού επιπέδου, καθώς αυτό είναι πιθανό να οδηγήσει στην αναποτελεσματική εκτέλεσή του. Για αυτό τον λόγο, η συγγραφή ενός shellcode μπορεί να αποδειχθεί ως μια διαδικασία δύσκολη και κοπιώδες.

Ο όρος shellcode προέρχεται από τον αρχικό σκοπό παραγωγής των shellcodes. Αποτελούσαν το τμήμα της επίθεσης που ήταν υπεύθυνο για την δημιουργία ενός shell το οποίο θα έφερε δικαιώματα διαχειριστή. Το συγκεκριμένο είδος shellcode παραμένει το πιο ευρέως χρησιμοποιούμενο, χωρίς όμως να είναι το μόνο, καθώς έχουν δημιουργηθεί shellcodes με μια πληθώρα διαφορετικών δυνατοτήτων. Η λογική που διέπει τον τρόπο χρήσης ενός shellcode είναι απλή και μπορεί να χωριστεί σε δύο διακριτά τμήματα. Θα πρέπει να δοθεί με κάποιον τρόπο το shellcode ως είσοδος στον ευάλωτο πρόγραμμα και στην συνέχεια να τροποποιηθεί η ροή εκτέλεσης του συγκεκριμένου προγράμματος έτσι ώστε να εκτελέσει το παρεχόμενο shellcode.

3.5.1 Δημιουργία Shellcode για Windows

Τα shellcodes για Windows διαφέρουν από τα shellcodes που είναι συμβατά με Unix συστήματα. Η κύρια διαφορά έγκειται στο γεγονός ότι, στο λειτουργικό σύστημα των Windows δεν υπάρχουν κλήσεις συστήματος (system calls) με ένα γνωστό API. Αντίθετα, κάθε διεργασία αποθηκεύει δείκτες εξωτερικών συναρτήσεων σε διάφορα σημεία της μνήμης. Στην προκειμένη περίπτωση, ο επιτιθέμενος, που θέλει να δημιουργήσει το shellcode, δεν γνωρίζει τις διευθύνσεις που αποθηκεύονται οι δείκτες αυτοί.

Για την εξυπηρέτηση της λειτουργίας τους, τα Windows χρησιμοποιούν ένα σύνολο δυναμικών βιβλιοθηκών οι οποίες είναι γνωστές ως Dynamic Link Libraries (DLLs). Τα DLLs παρέχουν έναν τρόπο στις διεργασίες ώστε να καλούν συναρτήσεις που δεν είναι μέρος του δικούς τους εκτελέσιμου κώδικα. Ο εκτελέσιμος κώδικας της καλούμενης συνάρτησης περιέχεται σε ένα DLL, το οποίο DLL περιλαμβάνει μία ή περισσότερες μεταγλωττισμένες συναρτήσεις. Το Windows API υλοποιείται ως ένα σύνολο από DLLs, έτσι κάθε διεργασία που χρησιμοποιεί το Win32 API εκμεταλλεύεται τις δυνατότητες της δυναμικής διασύνδεσης. Όπως κάθε μοντέρνο λειτουργικό σύστημα, έτσι και τα Windows, χρησιμοποιούν μία κατάλληλη δομή αρχείου που φορτώνεται κατά τον χρόνο εκτέλεσης, παρέχοντας την λειτουργία των διαμοιραζόμενων βιβλιοθηκών.

Κάθε DLL είναι ένα PE-COFF αρχείο ή αλλιώς ένα Portable Executable (PE). Τα PE αρχεία περιέχουν έναν πίνακα (export table) που αναφέρει τις συναρτήσεις που εξάγει το

συγκεκριμένο DLL καθώς επίσης, καταδεικνύουν και την διεύθυνση μνήμης που μπορεί να βρεθεί η κάθε συνάρτηση.

Η πληροφορία κλειδί για την συγγραφή αξιόπιστων και επαναχρησιμοποιήσιμων shellcodes, είναι ότι τα Windows αποθηκεύουν ένα δείκτη στο Process Environment Block (PEB), ο οποίος συναντάται σε γνωστή τοποθεσία και πιο συγκεκριμένα στην FS: [0x30]. Αν προστεθεί στην διεύθυνση αυτή, το 0xc, τότε έχουμε τον δείκτη που αναφέρεται στην λίστα η οποία περιέχει τα modules (DLLs) που έχουν φορτωθεί. Σε αυτό το σημείο υπάρχει η δυνατότητα, για παράδειγμα, της προσπέλασης της διασυνδεδεμένης λίστας έτσι ώστε να βρεθεί το kernel32.dll. Έπειτα, μπορούν να εντοπιστούν οι συναρτήσεις LoadLibraryA() και GetProcAddress() του kernel32.dll, οι οποίες επιτρέπουν την φόρτωση οποιουδήποτε DLL καθώς και την εύρεση της διεύθυνσης κάθε συνάρτησης που περιλαμβάνεται σε αυτά τα DLLs.

3.5.1.1 Αλληλεπίδραση shellcode με Process Environment Block (PEB)

Στα πλαίσια του λειτουργικού συστήματος Windows, το Process Environment Block (PEB) είναι μία δομή διαθέσιμη για κάθε διεργασία που συναντάται σε μια προκαθορισμένη διεύθυνση μνήμης. Περιέχει χρήσιμες πληροφορίες σχετικά με την διεργασία, όπως είναι: η διεύθυνση μνήμης στην οποία έχει φορτωθεί το εκτελέσιμο και η λίστα με τα διαθέσιμα DLLs.

Το PEB υπό κανονικές συνθήκες θα πρέπει να χρησιμοποιείται μόνο από το ίδιο το λειτουργικό σύστημα. Ενδέχεται να υπάρχουν τροποποιήσεις της δομής αυτής, ανάμεσα στις διαφορετικές εκδόσεις Windows, όμως παρόλα αυτά, τα βασικά χαρακτηριστικά της παραμένουν αναλλοίωτα.

Τα DLLs εξαιτίας του μηχανισμού ασφάλειας Address Space Layout Randomization (ASLR) φορτώνονται σε διαφορετικές θέσεις μνήμης κάθε φορά και συνεπώς, για να είναι ένα shellcode λειτουργικό, θα πρέπει να αποφεύγεται η χρησιμοποίηση προκαθορισμένων διευθύνσεων μνήμης από αυτό. Ωστόσο, μπορεί να αξιοποιηθεί η δομή PEB έτσι ώστε να βρεθούν οι διευθύνσεις μνήμης στις οποίες έχουν φορτωθεί τα διάφορα DLLs. Το χαρακτηριστικό που καθιστά το PEB αξιοποιήσιμο για την εξαγωγή πληροφοριών, είναι ότι, βρίσκεται σε συγκεκριμένη θέση της μνήμης.

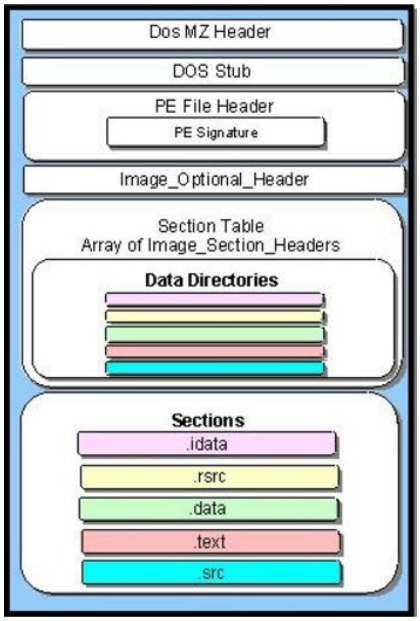
Τα ακριβή βήματα για την εύρεση της διεύθυνσης των DLLs είναι τα εξής:

1. Ανάγνωση της δομής PEB.
2. Μετάβαση στο offset 0xC, όπου βρίσκεται ο Ldr δείκτης.
3. Μετάβαση στο offset 0x14, όπου βρίσκεται το InMemoryOrderModuleList πεδίο.

Σε αυτό το σημείο έχει εντοπιστεί μία διπλή διασυνδεδεμένη λίστα (InMemoryOrderModuleList), η οποία διατηρεί τις σχετικές πληροφορίες των διαθέσιμων DLLs.

3.5.1.2 Δομή Portable Executable (PE) και αλληλεπίδραση με shellcode

Το portable executable συνθέτει μία συγκεκριμένη δομή η οποία συναντάται στο λειτουργικό σύστημα Windows τόσο στην έκδοση των 32-bit όσο και σε αυτήν των 64-bit. Πιο συγκεκριμένα, η δομή αρχείου που ορίζει το portable executable, χρησιμοποιείται από τα εκτελέσιμα και τις δυναμικές βιβλιοθήκες (DLLs), και περιγράφει τα περιεχόμενα των αρχείων αυτών. Στο σημείο αυτό, θα περιγράψουμε τα τμήματα του portable executable με τα οποία αλληλεπιδρά ένα shellcode. Η παρούσα ανάλυση θα βοηθήσει στην κατανόηση μετέπειτα ενοτήτων. Στο παρακάτω στιγμιότυπο απεικονίζονται τα βασικά τμήματα ενός PE αρχείου.



Εικόνα 2: Βασική δομή ενός portable executable.

Όπως φαίνεται και στην εικόνα 2, το PE αρχείο αποτελείται από:

- Την DOS επικεφαλίδα.
- Το DOS stub, το οποίο είναι υπεύθυνο για την εκτύπωση του μηνύματος “This program cannot be run in DOS mode”.
- Το τμήμα PE header, το οποίο περιέχει χρήσιμες πληροφορίες.
- Το section table.
- Τα διάφορα λοιπά τμήματα τα οποία περιλαμβάνουν τον κώδικα και τα δεδομένα του προγράμματος.

Η δομή ενός portable executable είναι αρκετά πολύπλοκη, όμως στο πλαίσιο που το μελετάμε, θα πρέπει να γίνει κατανοητός μόνο ο τρόπος που ένα shellcode προσπελαύνει τις κατάλληλες PE επικεφαλίδες έτσι ώστε να εντοπίσει τις εξαγόμενες συναρτήσεις. Όλα τα PE αρχεία (exe και DLL) ξεκινούν με την DOS επικεφαλίδα και τα δύο πρώτα bytes είναι οι χαρακτήρες “MZ”, οι οποίοι αποτελούν το e_magic πεδίο. Από τα πεδία της DOS επικεφαλίδας, ιδιαίτερα σημαντικό είναι το e_lfanew πεδίο το οποίο συναντάται στο offset 0x3C και δείχνει στην αρχή του τμήματος PE header. Στο συγκεκριμένο τμήμα, θα πρέπει να

αναζητηθεί το πεδίο με την ονομασία Optional Header. Τα πιο σημαντικά πεδία της δομής Optional Header είναι τα εξής:

- AddressOfEntryPoint, όπου το exe/DLL ξεκινά να εκτελεί κώδικα.
- ImageBase, δείχνει στην διεύθυνση μνήμης που θα έπρεπε να φορτωθεί το DLL εάν αυτό είναι δυνατό.
- DataDirectory, περιέχει για τις συναρτήσεις που εξάγει το συγκεκριμένο DLL.

Το πεδίο που μας ενδιαφέρει είναι το DataDirectory, καθώς μέσω αυτού μπορούν να βρεθούν οι εξαγώμενες συναρτήσεις. Αυτός είναι και ο τρόπος με τον οποίο λειτουργούν τα DLLs αφού περιέχουν συναρτήσεις οι οποίες χρησιμοποιούνται από άλλες εφαρμογές, με την προϋπόθεση ότι προτού γίνει οποιαδήποτε κλήση συνάρτησης, η εφαρμογή θα πρέπει να έχει φορτώσει στην μνήμη το κατάλληλο DLL.

Το πεδίο DataDirectory είναι ένας πίνακας που συνθέτεται από δομές που ονομάζονται IMAGE_DATA_DIRECTORY. Η σχετική απόσταση από την αρχή του PE header έως τον DataDirectory πίνακα είναι 120 (0x78) bytes. Έτσι, στο offset 0x78 μπορεί να βρεθεί η εικονική διεύθυνση της δομής (IMAGE_EXPORT_DIRECTORY), η οποία είναι υπεύθυνη για την διατήρηση των πληροφοριών που αφορούν τις συναρτήσεις που εξάγει το συγκεκριμένο DLL. Συνδυάζοντας τα πεδία AddressOfFunctions, AddressOfNames και AddressOfNameOrdinals μπορεί να βρεθεί η διεύθυνση στην οποία έχει φορτωθεί η εκάστοτε συνάρτηση.

4. Διαδικασία μετατροπής shellcode σε μορφή ROP

Στο παρόν κεφάλαιο θα αναλυθεί η μεθοδολογία μετατροπής ενός shellcode σε μορφή Return-Oriented Programming (ROP) καθώς και ο τρόπος αντίστροφης ανάλυσης των εντολών assembly, από τις οποίες αποτελείται ένα shellcode. Για την παραγωγή του τελικού εκτελέσιμου αρχείου (exe file), δημιουργήθηκε ένα αρχείο αποτελούμενο από εντολές της γλώσσας προγραμματισμού C, στο οποίο ενσωματώθηκαν εν συνεχεία, οι εντολές assembly του shellcode σε μορφή ROP. Θα περιγράψουμε την δομή του συγκεκριμένου αρχείου, τις σημαντικότερες εντολές C που περιέχει καθώς και τον τρόπο λειτουργίας του. Επιπρόσθετα, θα παρουσιάσουμε τα πιο ενδιαφέροντα τμήματα της μετατροπής σε ROP, των δύο shellcodes με τα οποία ασχοληθήκαμε στα πλαίσια αυτής της διπλωματικής εργασίας. Τέλος, θα παρουσιάσουμε τους δείκτες αποφυγής των εκτελέσιμων σε σχέση με τα υπάρχοντα διαθέσιμα antivirus.

4.1 Γενική δομή αρχείου C

Στην ενότητα αυτή, θα περιγράψουμε την δομή του αρχείου C που δημιουργήθηκε για την ενσωμάτωση των εντολών assembly του shellcode, με στόχο την παραγωγή του μη ανιχνεύσιμου εκτελέσιμου. Οι εντολές assembly που περιλαμβάνονται στο αρχείο αυτό, πρέπει προηγουμένως να έχουν μετατραπεί σε μια ROP αλυσίδα, έτσι ώστε να αλλάζει σημαντικά η υπογραφή (signature) του shellcode και τελικά, να αποφεύγεται η ανίχνευση του μολυσμένου εκτελέσιμου από τα antivirus. Παρακάτω, παραθέτουμε τις σημαντικότερες εντολές του αρχείου C και προχωράμε στην περιγραφή των τμημάτων του καθώς και του τρόπου λειτουργίας του.

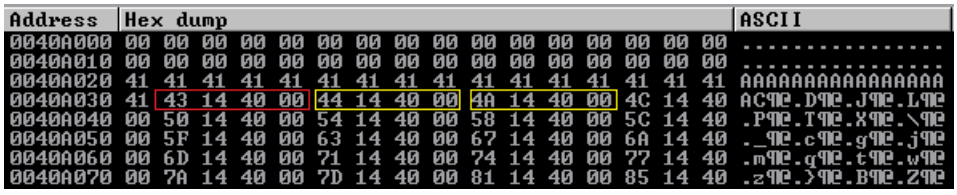
```
#define MAX_OP 100000000
void getprocaddr(){
        asm( Shellcode σε μορφή ROP );
}

int main(int argc, char * argv[])
{
        int cpt = 0;
        int i = 0;
        for(i =0; i < MAX_OP; i ++)  {
                cpt++;
        }
        if (cpt == MAX_OP)  {
                char buflarge[] = { Fuzzing + Διεύθυνση shellcode + Διευθύνσεις επιστροφής για την κλήση των gadgets };
                test(buflarge);
        }
}
void test(const char* buflarge){
        char buf[5];
        memcpy(buf, buflarge, 580);
        printf(buf);
}
```

4.1.1 Αποθήκευση εντολών assembly στην μνήμη του εκτελέσιμου

Οι εντολές assembly του shellcode, θα πρέπει με κάποιον τρόπο να εισαχθούν στην μνήμη του εκτελέσιμου και στην συνέχεια να δοθεί ο έλεγχος ροής εκτέλεσης στο συγκεκριμένο τμήμα της μνήμης έτσι ώστε να εκτελέσει το shellcode την λειτουργία του. Χρησιμοποιούμε την εντολή “asm” και τοποθετούμε την ισοδύναμη rop αλυσίδα του shellcode, στο σημείο αυτό. Μετά την μεταγλώττιση του αρχείου C, όλες οι εντολές assembly που περιλαμβάνονται στο εσωτερικό της εντολής “asm”, μπορούν να εντοπιστούν στην μνήμη του παραγόμενου εκτελέσιμου και πιο συγκεκριμένα στο τμήμα text.

4.1.2 Δημιουργία βοηθητικού buffer overflow

Προκειμένου να μεταφέρουμε την ροή εκτέλεσης του προγράμματος στο σημείο της μνήμης που έχουν αποθηκευτεί οι εντολές assembly του shellcode, δημιουργούμε ένα buffer overflow. Ορίζουμε έναν buffer με δυνατότητα αποθήκευσης δεδομένων που ανέρχεται μόλις στα 5 bytes και στην συνέχεια, προκαλούμε την υπερχείλισή του καλώντας την συνάρτηση “memcpy”, εγγράφοντας περισσότερα δεδομένα από αυτά που μπορεί να χωρέσει ο συγκεκριμένος buffer. Επανεγγράφουμε την διεύθυνση επιστροφής που υπάρχει στην στοίβα του προγράμματος, με την διεύθυνση που εντοπίζεται η πρώτη εντολή assembly του shellcode, προκαλώντας έτσι την κλήση της ROP αλυσίδας εντολών (κλήση πρώτου gadget) που έχουμε δημιουργήσει. Οι εντολές της ROP αλυσίδας, τελειώνουν με την εντολή επιστροφής ret. Ως εκ τούτου, για να λειτουργήσει σωστά το shellcode σε ROP μορφή, θα πρέπει να εισάγουμε στην στοίβα του προγράμματος, εκτός από την πρώτη διεύθυνση του shellcode, και τις απαραίτητες μετέπειτα διευθύνσεις. Να σημειώσουμε, ότι η εντολή ret αντιστοιχεί με “pop eip”. Έτσι, κάθε εκτέλεση της εντολής ret θα πρέπει να έχει ως αποτέλεσμα την τοποθέτηση της σωστής τιμής στον καταχωρητή eip, δηλαδή την διεύθυνση του επόμενου gadget. Στο παρακάτω στιγμιότυπο, απεικονίζεται η κατάσταση της στοίβας του προγράμματος μετά το buffer overflow.



Εικόνα 3: Στοίβα του προγράμματος μετά την υπερχείλιση του buffer.

Οι δεκαεξαδικοί χαρακτήρες “0x41” εισάγονται έτσι ώστε να καλυφθεί η σχετική απόσταση μέχρι την διεύθυνση επιστροφής. Με κόκκινο χρώμα, φαίνεται η διεύθυνση επιστροφής η οποία αντικαθίσταται από την διεύθυνση του text τομέα που εντοπίζεται το shellcode, ενώ με κίτρινο χρώμα, ακολουθούν οι διευθύνσεις των πρώτων gadgets της ROP αλυσίδας.

4.1.3 Αποφυγή δυναμικής ανάλυσης από τα antivirus

Τα περισσότερα antivirus, έχουν δυνατότητες δυναμικής ανάλυσης των αρχείων έτσι ώστε να είναι σε θέση να εξακριβώσουν κατά πόσο ένα αρχείο μπορεί να είναι μολυσμένο. Οι μηχανές που έχουν τα antivirus για την διεξαγωγή της δυναμικής ανάλυσης, προσπαθούν είτε να εντοπίσουν συγκεκριμένες κλήσεις API, είτε εκτελούν το πρόγραμμα που

διερευνάται, στα πλαίσια ενός περιβάλλοντος προσομοίωσης [8]. Εκτελώντας το πρόγραμμα σε ένα προστατευμένο περιβάλλον, τα antivirus μπορούν να εξάγουν συμπεράσματα σχετικά με την συμπεριφορά του προγράμματος κατά την διάρκεια εκτέλεσής του. Προκειμένου να αποφύγουμε την δυναμική ανάλυση των antivirus, εισάγαμε ένα μετρητή ο οποίος αυξάνεται επαναληπτικά έως ότου φτάσει στην τιμή "100000000". Αφού ολοκληρωθεί η αύξηση του μετρητή, ξεκινάνε να εκτελούνται οι πραγματικά χρήσιμες εργασίες, δηλαδή η υπερχείλιση του buffer και στην συνέχεια, η κλήση των εντολών assembly του shellcode. Η απλή αυτή τεχνική είναι αποτελεσματική καθώς τα antivirus εκτελούν το δυνητικά μολυσμένο πρόγραμμα στο προστατευμένο περιβάλλον που υλοποιούν, για σύντομο χρονικό διάστημα.

4.2 Αντίστροφη ανάλυση των x86 εντολών μηχανής

Η αντίστροφη ανάλυση των εντολών assembly από τις οποίες αποτελείται το shellcode, είναι ζωτικής σημασίας διαδικασία για την αντικατάσταση των εντολών με ισοδύναμες και την μετέπειτα μετατροπή τους σε ROP μορφή. Ιδιαίτερα σημαντική πληροφορία είναι ποιοι καταχωρητές διαβάζονται ή εγγράφονται κατά την διαδικασία εκτέλεσης κάθε εντολής, καθώς επίσης και ποιοι καταχωρητές είναι ελεύθεροι προς τροποποίηση. Με βάση τα αποτελέσματα της ανάλυσης αυτής, μπορεί να γίνει η αντικατάσταση καταχωρητών με άλλους καταχωρητές, χωρίς να διαταράσσεται η ομαλή λειτουργία του shellcode. Η πιθανή σημασιολογία και κατάσταση ενός καταχωρητή, στα πλαίσια εκτέλεσης κάθε εντολής, απεικονίζεται στον παρακάτω πίνακα.

Κατάσταση καταχωρητή	Επεξήγηση
READS(I, X)	Η εντολή I διαβάζει την τιμή του καταχωρητή X
WRITES(I, X)	Η εντολή I εγγράφει/τροποποιεί την τιμή του καταχωρητή X
ACCESSES(I, X)	READS(I, X) V WRITES(I,X)
SETS(I, X)	WRITES(I, X) ∧ ¬READS(I, X): Η εντολή I θέτει την τιμή του καταχωρητή X χωρίς γνωρίζει την προηγούμενη τιμή του
FREE(I, X)	Ο καταχωρητής X είναι ελεύθερος κατά την εκτέλεση της εντολής I, και έτσι οποιαδήποτε τροποποίησή του δεν θα αλλάξει την ροή εκτέλεσης ή το αποτέλεσμα της εκτέλεσης

Πίνακας 1: Πιθανή κατάσταση των καταχωρητών στο πλαίσιο εκτέλεσης των εντολών.

4.2.1 Χειρισμός εντολών που χρησιμοποιούν MOD/REG/RM ή SIB διευθυνσιοδότηση

Οι εντολές που χρησιμοποιούν τον έμμεσο τρόπο διευθυνσιοδότησης MOD/REG/RM ή το σχήμα διευθυνσιοδότησης Scaled Index Byte (SIB), απαιτούν ιδιαίτερο χειρισμό προκειμένου να αντικατασταθούν με ισοδύναμες εντολές. Στις περισσότερες των περιπτώσεων, οι εντολές αυτές διαβάζουν πολλούς καταχωρητές γενικού σκοπού ταυτόχρονα, περιορίζοντας σημαντικά το εύρος των ελεύθερων καταχωρητών. Για παράδειγμα, η εντολή: move eax, [ebx+ecx*2] μπορεί να αντικατασταθεί από τις παρακάτω εντολές.

```
[1] sal ecx, 1
[2] add ecx, ebx
[3] mov eax, [ecx]
```

Για να μην διαταραχθεί η ροή εκτέλεσης του shellcode, απαραίτητη προϋπόθεση είναι να ισχύει FREE(1, ecx), δηλαδή ο καταχωρητής ecx να μην είναι δεσμευμένος. Εναλλακτικά θα μπορούσε να πραγματοποιηθεί αντικατάσταση με τις παρακάτω εντολές.

```
[1] mov eax, ecx
[2] sal eax, 1
[3] add eax, ebx
[4] mov eax, [eax]
```

Στην περίπτωση αυτή απαιτείται SETS(1, eax). Αν καμία από αυτές τις δύο αντικαταστάσεις δεν είναι δυνατή, τότε θα πρέπει να αναζητηθεί κάποιος άλλος ελεύθερος καταχωρητής.

4.3 Μετατροπή του shellcode σε μορφή ROP

Η μετατροπή του shellcode στο return-oriented ισοδύναμο, περιλαμβάνει την δημιουργία μιας αλυσίδας αποτελούμενης από gadgets τα οποία θα πρέπει να εκτελούνται σειριακά. Για την σωστή εκτέλεση όλων των gadgets θα πρέπει να εισαχθούν στην στοίβα του προγράμματος οι διευθύνσεις που αντιστοιχούν στην πρώτη εντολή κάθε gadget. Στα πλαίσια της συγκεκριμένης διπλωματικής εργασίας, δεν υπάρχει η ανάγκη για εύρεση των gadgets καθώς όπως περιγράφηκε σε προηγούμενη ενότητα, έχουμε την δυνατότητα να εισάγουμε τις εντολές assembly που επιθυμούμε και συνεπώς να δημιουργούμε τα απαραίτητα gadgets.

4.3.1 Μετατροπή "απλών" εντολών σε μορφή ROP

Στην κατηγορία των "απλών" εντολών κατατάσσουμε όλες τις εντολές που δεν αλλάζουν την κατάσταση της στοίβας του προγράμματος. Η μετατροπή μίας τέτοιας εντολής σε ROP, είναι σχετικά απλή, καθώς μετά την ολοκλήρωση της αντικατάστασης της εντολής με κάποιες ισοδύναμες, το μόνο που χρειάζεται είναι να προσθέσουμε μετά την τελευταία εντολή, την εντολή επιστροφής ret. Με αυτόν τον τρόπο έχουμε δημιουργήσει ένα gadget. Είναι απαραίτητο να εισάγουμε στην στοίβα του προγράμματος την διεύθυνση του επόμενου gadget,η οποία μπορεί να βρεθεί με την βοήθεια ενός debugger, έτσι ώστε να λειτουργήσει σωστά η εντολή ret.

4.3.2 Μετατροπή εντολών που τροποποιούν την κατάσταση της στοίβας σε ROP

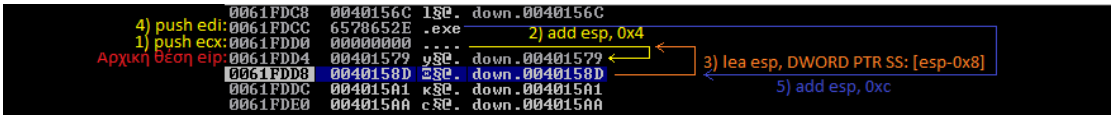
Εντάσσουμε στην κατηγορία των εντολών που τροποποιούν την κατάσταση της στοίβας του προγράμματος, κυρίως τις εντολές push. Για να μετατρέψουμε σε ROP μορφή τέτοιες εντολές χρησιμοποιούμε την εντολή add για την αύξηση της τιμής του esp και την εντολή lea για την μείωση της τιμής του esp. Είναι απαραίτητο να προσαρμόζουμε την τιμή του esp κάθε φορά που θέλουμε να εισάγουμε κάποια τιμή στην στοίβα.

Για την καλύτερη κατανόηση της συγκεκριμένη διαδικασίας θα χρησιμοποιήσουμε ένα παράδειγμα. Έστω ότι, πρέπει να εισαχθούν στην κορυφή της στοίβας οι τιμές των καταχωρητών ecx και edi. Οι εντολές assembly που συνθέτουν τα κατάλληλα gadgets για την ολοκλήρωση της εισαγωγής των καταχωρητών ecx και edi, είναι οι εξής:

```
0x00401574: push ecx
                add esp, 0x4
```

```
                ret

0x00401579: lea esp, DWORD PTR SS: [esp-0x8]
                push edi
                add esp, 0xc
                ret

0x0040158d: διεύθυνση επόμενου gadget
```

Η κατάσταση της στοίβας μετά την ολοκλήρωση των εντολών που περιέχουν τα δύο gadgets, απεικονίζεται στο παρακάτω στιγμιότυπο.



Εικόνα 4: Τρόπος λειτουργίας gadgets που εισάγουν τιμές στην στοίβα του προγράμματος.

Η εκτέλεση του ret που περιέχεται στο δεύτερο gadget θα έχει ως αποτέλεσμα την εισαγωγή της τιμής "0x0040158d" στον καταχωρητή eip και ως εκ τούτου την κλήση του επόμενου gadget της αλυσίδας.

4.3.3 Μετατροπή εντολών που περιέχονται σε επαναληπτική δομή σε ROP

Για τις εντολές που περιέχονται σε κάποιον επαναληπτικό βρόγχο, δεν μπορούμε να προβλέψουμε εκ των προτέρων, πόσες φορές θα εκτελεστούν. Οπότε, δεν είναι σωστή τακτική να εισάγουμε στην στοίβα του προγράμματος μέσω του buffer overflow, την διεύθυνση των επόμενων gadgets, καθώς αυτό είναι πιθανό να έχει ως αποτέλεσμα την μη σωστή εκτέλεση του shellcode. Μία αποτελεσματική λύση είναι, μετά την εντολή που θέλουμε να μετατρέψουμε σε ROP, να προσθέτουμε την εντολή push με την διεύθυνση του επόμενου gadget, ακολουθούμενη από την εντολή επιστροφής ret. Έστω ότι θέλουμε να μετατρέψουμε σε ROP, τον παρακάτω επαναληπτικό βρόγχο:

```
[address1] pointer: inc eax
[address2]          mov edx, ebx
[address3] loop pointer
```

Μπορούμε να επιτύχουμε την μετατροπή σε ROP, ως εξής:

```
[address1]pointer: inc eax
                   push address2'
                   ret
[address2']        mov edx, ebx
                   push address3'
                   ret
[address3']loop pointer
```

4.4 "Allwin URLDownloadToFile + WinExec + ExitProcess" shellcode

Το πρώτο shellcode το οποίο μετατρέψαμε σε ROP μορφή, μπορεί να βρεθεί στο Διαδίκτυο ως "Allwin URLDownloadToFile + WinExec + ExitProcess" shellcode [9]. Το συγκεκριμένο

shellcode, εκτελεί την συνάρτηση URLDownloadToFile έτσι ώστε να κατεβάσει ένα αρχείο από ένα προκαθορισμένο URL και στην συνέχεια, το εκτελεί χρησιμοποιώντας την συνάρτηση WinExec. Τέλος, τερματίζει την διεργασία που είναι υπεύθυνη για την εκτέλεση του shellcode αξιοποιώντας την συνάρτηση ExitProcess.

4.4.1 Ανάλυση εντολών assembly του shellcode

Σε αυτήν την ενότητα, θα αναλύσουμε περίπου το 30% των εντολών assembly από τις οποίες αποτελείται το συγκεκριμένο shellcode, καθώς παρουσιάζει ιδιαίτερο ενδιαφέρον ο τρόπος με τον οποίο εντοπίζονται οι διευθύνσεις των απαραίτητων συναρτήσεων μέσα στα αντίστοιχα DLLs.

```
[1] xor ecx, ecx
[2] mov eax, DWORD PTR FS:[ecx+0x30]
[3] mov eax, DWORD PTR DS:[eax+0xc]
[4] mov esi, DWORD PTR DS:[eax+0x14]
[5] lods DWORD PTR DS:[esi]
[6] xchg eax, esi
[7] lods DWORD PTR DS:[esi]
[8] mov ebx, DWORD PTR DS:[eax+0x10]
```

Η εντολή [1] μηδενίζει την τιμή του καταχωρητή ecx. Με τις εντολές [2], [3], [4] εντοπίζουμε το Process Environment Block (PEB), το δείκτη Ldr και το πεδίο InMemoryOrderModuleList, αντίστοιχα. Η εντολή lods, ακολουθεί το δείκτη που ορίζεται από τον καταχωρητή esi και αποθηκεύει το αποτέλεσμα στον καταχωρητή eax. Αυτό σημαίνει ότι με την εκτέλεση της εντολής [5] έχουμε το δεύτερο module (ntdll.dll) στον καταχωρητή eax. Αφού ολοκληρωθεί, η εκτέλεση της εντολής [7], έχει εντοπιστεί το τρίτο module (kernel32.dll), το οποίο ήταν και ο στόχος της αναζήτησης. Η τιμή του καταχωρητή eax δείχνει στο kernel32.dll και προσθέτοντας 0x10 bytes μπορεί να βρεθεί η διεύθυνση μνήμης στην οποία έχει φορτωθεί το συγκεκριμένο module (εντολή [8]).

```
[9] mov edx, DWORD PTR DS:[ebx+0x3c]
[10] add edx, ebx
[11] mov edx, DWORD PTR DS:[edx+0x78]
[12] add edx, ebx
[13] mov esi, DWORD PTR DS:[edx+0x20]
[14] add esi, ebx
[15] xor ecx, ecx
```

Με την εντολή [9], εντοπίζεται ο δείκτης e_lfanew και προσθέτοντας την τιμή του δείκτη αυτού στην διεύθυνση βάσης που βρήκαμε προηγουμένως (εντολή [10]), έχουμε την διεύθυνση της PE επικεφαλίδας. Στο offset 0x78 της PE επικεφαλίδας, μπορεί να βρεθεί ο DataDirectory πίνακας. Με βάση αυτήν την πληροφορία, οι εντολές [11] και [12] είναι υπεύθυνες για τον εντοπισμό του πίνακα (IMAGE_EXPORT_DIRECTORY) που δηλώνει τις συναρτήσεις που εξάγει το module kernel32.dll. Στο offset 0x20 του πίνακα αυτού, μπορεί να βρεθεί ο δείκτης που αναφέρεται στην δομή AddressOfNames (εντολή [13]), η οποία ουσιαστικά είναι ένα πίνακας αποτελούμενος από δείκτες. Κάθε 4 bytes του πίνακα, αντιπροσωπεύουν έναν δείκτη που αναφέρεται στο όνομα μίας από τις συναρτήσεις που εξάγει το kernel32.dll. Η διεύθυνση του πίνακα AddressOfNames αποθηκεύεται στον καταχωρητή esi με την εντολή [14].

```
[16] inc ecx
[17] lods DWORD PTR DS:[ESI]
[18] add eax, ebx
[19] cmp DWORD PTR DS:[eax], 0x50746547
[20] jnz short [16]
[21] cmp DOWRD PTR DS:[eax+0x4], 0x41636f72
[22] jnz short [16]
[23] cmp DWORD PTR DS:[eax+0x8], 0x65726565
[24] jnz short [16]
```

Οι παραπάνω εντολές εντοπίζουν τη θέση της συνάρτησης GetProcAddress στον πίνακα AddressOfNames με βάση το όνομά της και αποθηκεύουν το αποτέλεσμα στον καταχωρητή ecx. Με την εντολή [17] προσπελαύνουμε τα στοιχεία του πίνακα αυτού, ενώ με την εντολή cmp γίνονται οι απαραίτητες συγκρίσεις. Οι ascii τιμές που συγκρίνονται κάθε φορά, αντιστοιχίζονται στους εξής χαρακτήρες:

- 0x50746547: GetP
- 0x41636f72: rocA
- 0x65726565: ddre

Αν κάποια από τις συγκρίσεις αποτύχει, τότε η διαδικασία επαναλαμβάνεται εκτελώντας ξανά την εντολή [16].

```
[25] mov esi, DWORD PTR DS:[edx+0x24]
[26] add esi, ebx
[27] mov cx, WORD PTR DS:[esi+ecx*2]
[28] dec ecx
[29] mov esi, DWORD PTR DS:[ecx+0x1c]
[30] add esi, ebx
[31] mov edx, DWORD PTR DS:[esi+ecx*4]
[32] add edx, ebx
```

Ο καταχωρητής edx δείχνει στον πίνακα IMAGE_EXPORT_DIRECTORY και έτσι, προσθέτοντας το offset 0x24, μπορούμε να βρούμε την δομή AdressOfNameOrdinals (εντολή [25]). Με την εντολή [27] βρίσκουμε την θέση της συνάρτησης GetProcAddress, μέσα στον επόμενο πίνακα που θα χρησιμοποιηθεί, ο οποίος ονομάζεται AddressOfFunctions και το αποτέλεσμα αποθηκεύεται στον καταχωρητή ecx. Είναι απαραίτητη η μείωση του ecx (εντολή [28]) καθώς ο πίνακας AddressOfNameOrdinals ξεκινάει από το μηδέν. Οι εντολές [29] και [30] είναι υπεύθυνες για τον εντοπισμό της διεύθυνσης του πίνακα AddressOfFunctions με την διεύθυνση να αποθηκεύεται στον καταχωρητή esi. Οι τελευταίες εντολές βρίσκουν την διεύθυνση της συνάρτησης GetProcAddress με βάση τον πίνακα AddressOfFunctions και αποθηκεύουν το αποτέλεσμα στον καταχωρητή edx.

Σε αυτό το σημείο θα σταματήσουμε με την ανάλυση των εντολών assembly του shellcode καθώς έχουμε αναλύσει ήδη τα πιο αξιοσημείωτα τμήματά του. Ο μετέπειτα τρόπος λειτουργίας του είναι σχετικά απλός, καθώς αφού έχει βρεθεί η διεύθυνση της συνάρτησης GetProcAddress, μπορεί να χρησιμοποιηθεί για την εύρεση και της συνάρτησης LoadLibrary, η οποία περιέχεται επίσης στο kernel32.dll. Η συνάρτηση LoadLibrary μπορεί να αξιοποιηθεί για την φόρτωση οποιουδήποτε DLL, το οποίο είναι απαραίτητο για την

λειτουργία του shellcode. Ουσιαστικά, χρησιμοποιούνται επαναληπτικά οι συναρτήσεις GetProcAddress και LoadLibrary.

4.4.2 Μετατροπή σε μορφή ROP

Όπως φαίνεται από την ανάλυση της προηγούμενης ενότητας, στο πρώτο τμήμα του shellcode, οι εντολές εκτελούνται σειριακά, χωρίς να υπάρχουν επαναληπτικοί βρόγχοι. Οπότε σε αυτό το τμήμα, προχωράμε όπου είναι εφικτό σε αντικαταστάσεις των εντολών με ισοδύναμες, και για την μετατροπή σε ROP εφαρμόζουμε την μεθοδολογία που περιγράφηκε στην ενότητα 4.3.1. Το υπόλοιπο τμήμα του shellcode αποτελείται από διαδοχικές κλήσεις συναρτήσεων. Δηλαδή, εκτελούνται εντολές εισαγωγής τιμών στην στοίβα του προγράμματος (εντολές push), οι οποίες τιμές αποτελούν τα ορίσματα της εκάστοτε συνάρτησης. Τέλος, ακολουθεί η κλήση της κατάλληλης συνάρτησης. Επειδή, το τμήμα αυτό περιέχει πολλές εντολές που έχουν ως αποτέλεσμα την τροποποίηση της στοίβας, εφαρμόζουμε την μεθοδολογία που περιγράφηκε στην ενότητα 4.3.2.

1ο τμήμα εντολών: εκτέλεση σειριακών εντολών **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.1**
2ο τμήμα εντολών: εκτέλεση εντολών push και κλήση συνάρτησης GetProcAddress **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.2**
3ο τμήμα εντολών: εκτέλεση εντολών push και κλήση συνάρτησης LoadLibrary **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.2**
4ο τμήμα εντολών: εκτέλεση εντολών push και κλήση συνάρτησης GetProcAddress **Μεθοδολογία μετατροπής σε ROP: ενότητα 34.3.2**

Πίνακας 2: Μεθοδολογία μετατροπής σε ROP για κάθε τμήμα του shellcode.

Στον παραπάνω πίνακα, δεν απεικονίζονται όλα τα τμήματα του shellcode καθώς ακολουθούν και άλλες κλήσεις συναρτήσεων. Ωστόσο, η μεθοδολογία μετατροπής των τμημάτων αυτών σε ROP μορφή, παραμένει η ίδια.

4.5 Reverse TCP Shell του Metasploit Framework

Το δεύτερο shellcode που μετατρέψαμε σε μορφή ROP, μπορεί να βρεθεί στα payloads που παρέχονται από το Metasploit Framework [10], ως windows/shell/reverse_tcp. Ειδικότερα, μετατρέψαμε σε ROP, το stager του συγκεκριμένου payload, το οποίο είναι το reverse_tcp. Με την εκτέλεσή του, δημιουργεί ένα shell ανάμεσα στον υπολογιστή του “θύματος” και του επιτιθέμενου, κατεβάζοντας στο τερματικό του “θύματος” το αντίστοιχο stage, δηλαδή το shell. Δεν θα προχωρήσουμε σε ανάλυση των εντολών assembly του συγκεκριμένου shellcode, αλλά θα περιγράψουμε την διαδικασία μετατροπής του σε μορφή ROP.

4.5.1 Μετατροπή σε μορφή ROP

Το πρώτο τμήμα του reverse tcp shell, αποτελείται από έναν επαναληπτικό βρόγχο. Επειδή, δεν μπορούμε να υπολογίσουμε πόσες φορές θα εκτελεστεί ο συγκεκριμένος επαναληπτικός βρόγχος, δεν είναι δυνατή η εκ των προτέρων εισαγωγή των διευθύνσεων των gadgets στην στοίβα του προγράμματος μέσω του buffer overflow. Για την μετατροπή αυτού του τμήματος εντολών σε μορφή ROP εφαρμόζουμε την μεθοδολογία που περιγράφηκε στην ενότητα 4.3.3. Τα υπόλοιπα τμήματα του shellcode, αποτελούνται από

εντολές εισαγωγής τιμών στην στοίβα του προγράμματος και μετέπειτα, κλήσεις των κατάλληλων συναρτήσεων. Για την μετατροπή των τμημάτων αυτών σε μορφή ROP, εφαρμόζουμε την μεθοδολογία που περιγράφηκε στην ενότητα 4.3.2.

1ο τμήμα εντολών: εκτέλεση επαναληπτικού βρόγχου **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.3**
2ο τμήμα εντολών: εκτέλεση εντολών push **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.2**
3ο τμήμα εντολών: εκτέλεση εντολών push **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.2**
4ο τμήμα εντολών: εκτέλεση εντολών push **Μεθοδολογία μετατροπής σε ROP: ενότητα 4.3.2**

Πίνακας 3: Μεθοδολογία μετατροπής σε ROP του Reverse TCP Shell.

Στον παραπάνω πίνακα, δεν απεικονίζονται όλα τα τμήματα του shellcode, ωστόσο, η μεθοδολογία μετατροπής των τμημάτων αυτών σε ROP μορφή, παραμένει η ίδια.

4.6 Μεταγλώττιση του shellcode

Αφού ολοκληρωθεί η διαδικασία μετατροπής των shellcode σε μορφή ROP, εξάγουμε το εκτελέσιμο αρχείο χρησιμοποιώντας δύο διαφορετικούς τρόπους μεταγλώττισης, οι οποίοι είναι οι εξής:

- Minimalist GNU for Windows (MinGW)
- Microsoft Visual Studio

Το MinGW παρέχει ένα μινιμαλιστικό περιβάλλον ανάπτυξης για Microsoft Windows εφαρμογές. Για την μεταγλώττιση του εκάστοτε αρχείου C που περιείχε το shellcode σε ROP μορφή, χρησιμοποιήθηκε η παρακάτω εντολή:

gcc –m32 –masm=intel shellcode.c –o result

Χρησιμοποιούμε την σημαία (flag) “–m32” έτσι ώστε να δημιουργήσουμε ένα 32-bit εκτελέσιμο ενώ χρησιμοποιούμε την σημαία “-masm=intel” για να δηλώσουμε ότι οι εντολές που περικλείονται μέσα στην εντολή asm του αρχείου C, έχουν συνταχθεί με βάση το Intel πρότυπο.

Το Visual Studio είναι ένα περιβάλλον ανάπτυξης εφαρμογών (IDE) το οποίο παρέχεται από την Microsoft. Υποστηρίζει πολλές διαφορετικές γλώσσες προγραμματισμού και περιλαμβάνει τον δικό του ενσωματωμένο assembler. Αυτό σημαίνει ότι, επιτρέπει στα προγράμματα που βασίζονται στις γλώσσες προγραμματισμού C και C++, να περιέχουν εντολές assembly, χωρίς να απαιτούνται επιπρόσθετα βήματα, όπως για παράδειγμα, η εγκατάσταση ενός αυτόνομου assembler όπως είναι ο Microsoft Macro Assembler (MASM). Μπορεί να χρησιμοποιηθεί η δήλωση “__asm”, στο τμήμα της οποίας μπορούν να ενσωματωθούν οι εντολές assembly του shellcode. Κάθε τέτοια δήλωση, προκαλεί μία κλήση στον inline assembler του Visual Studio. Προτού προχωρήσουμε στην μεταγλώττιση του κώδικα C, θα πρέπει να γίνουν ορισμένες αλλαγές στις ιδιότητες (properties) της εφαρμογής έτσι ώστε να λειτουργήσει με τον προβλεπόμενο τρόπο το παραγόμενο εκτελέσιμο.

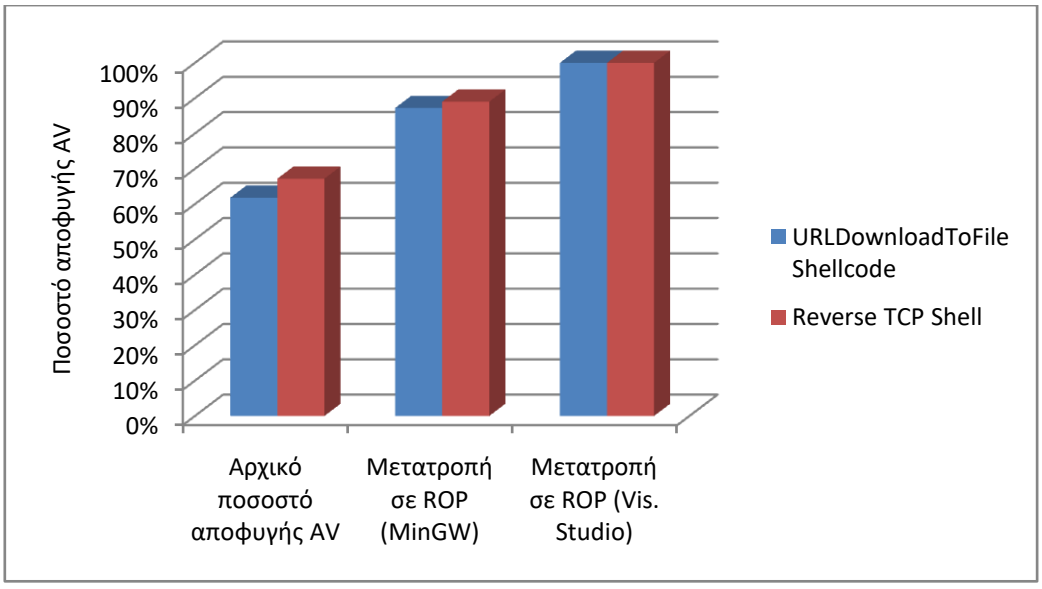
Όνομα ιδιότητας	Τιμή
Basic Runtime Checks	Default
Security Checks	Disable Security Checks(/GS-)
Randomized base Address	No(/DYNAMICBASE:NO)

Πίνακας 4: Αλλαγή ιδιοτήτων της εφαρμογής.

Οι παραπάνω αλλαγές απενεργοποιούν την τυχαιοποίηση της διεύθυνσης των εντολών κάθε φορά που γίνεται μεταγλώττιση του κώδικα C, όπως επίσης απενεργοποιούν τον αμυντικό μηχανισμό με την ονομασία Stack Smashing Protector (SSP). Η τελευταία αλλαγή είναι απαραίτητη καθώς σε διαφορετική περίπτωση δεν θα λειτουργήσει το εκτελέσιμο που παράγεται. Δημιουργούμε ένα εκτελέσιμο το οποίο περιέχει ένα ηθελημένο buffer overflow και για να γίνει επιτυχημένη υπερχείλιση της στοίβας του προγράμματος δεν θα πρέπει να είναι ενεργοποιημένος ο αντίστοιχος αμυντικός μηχανισμός.

4.7 Αποτελέσματα

Προκειμένου να αξιολογήσουμε την αποτελεσματικότητα των τεχνικών που χρησιμοποιήσαμε, εκμεταλλευτήκαμε τις δυνατότητες που παρέχει η ιστοσελίδα VirusTotal [11]. Πρόκειται για μια διαδικτυακή υπηρεσία που μπορεί να χρησιμοποιηθεί από τον καθένα, έτσι ώστε να διαπιστώσει εάν ένα αρχείο είναι μολυσμένο με κώδικα κακόβουλου χαρακτήρα. Αφού μετατρέψαμε σε μορφή ROP τα δύο shellcode, μεταγλωττίσαμε τα αντίστοιχα αρχεία C, χρησιμοποιώντας τόσο τον MinGW μεταγλωττιστή όσο και το Microsoft Visual Studio και στην συνέχεια, σαρώσαμε τα εκτελέσιμα αρχεία αξιοποιώντας την ιστοσελίδα VirusTotal. Στο παρακάτω διάγραμμα απεικονίζονται τα ποσοστά αποφυγής $(1 - \frac{positives}{total\ \ AVs})$ των μολυσμένων εκτελέσιμων από τα antivirus.



Εικόνα 5: Ποσοστά αποφυγής των antivirus για τα παραγόμενα εκτελέσιμα.

Το ποσοστό αποφυγής των AV, για το URLDownloadToFile Shellcode, αφού ολοκληρώθηκε η μετατροπή του σε μορφή ROP και μεταγλωττίστηκε με το MinGW, ανερχόταν στο 87,2%. Το αντίστοιχο ποσοστό για το Reverse TCP Shell ήταν το 89%. Και τα δύο shellcode ήταν μη ανιχνεύσιμα, δηλαδή το ποσοστό αποφυγής των AV ανερχόταν στο 100%, όταν για την

μεταγλώττιση των αντίστοιχων αρχείων C, χρησιμοποιήθηκε το Microsoft Visual Studio. Παρατηρούμε ότι, ανάλογα με τον μεταγλωττιστή που χρησιμοποιείται, προκύπτουν και διαφορετικά ποσοστά αποφυγής των AV.

5. Αυτοματοποίηση της μετατροπής shellcode σε μορφή ROP

Στα πλαίσια αυτής της διπλωματικής εργασίας, αναπτύξαμε ένα πρόγραμμα για την αυτοματοποίηση της διαδικασίας μετατροπής ενός shellcode σε μορφή ROP. Στο παρόν κεφάλαιο θα αναλυθούν οι δυνατότητες του συγκεκριμένου προγράμματος καθώς και ο τρόπος λειτουργίας του. Θα περιγράψουμε τον βασικό αλγόριθμο που χρησιμοποιεί για την δημιουργία της ROP αλυσίδας και πως είναι σε θέση να υπολογίζει με δυναμικό τρόπο τις διευθύνσεις των gadgets. Τέλος, θα περιγράψουμε τον τρόπο με τον οποίο αντικαθιστά εντολές assembly με ισοδύναμες, χωρίς να διαταράσσεται η συνολική λειτουργία του shellcode.

5.1 Δυνατότητες του script και είσοδοι

Το script που αναπτύξαμε, υλοποιήθηκε στην γλώσσα προγραμματισμού Python. Ο κύριος σκοπός της δημιουργίας του ήταν η όσο μεγαλύτερη αυτοματοποίηση της διαδικασίας που λάμβανε χώρα εντολή προς εντολή. Το συγκεκριμένο script, που φέρει την ονομασία rop_script.py, έχει την δυνατότητα δεδομένου ενός shellcode, να προχωράει στο disassembly των αντίστοιχων machine codes και στην συνέχεια, σε αντικατάσταση των εντολών με ισοδύναμες καθώς και στην δημιουργία μιας αλυσίδας από gadgets, τα οποία gadgets περιέχουν τις εντολές assembly του shellcode. Για την εκτέλεσή του, θα πρέπει να χρησιμοποιηθεί το Command Prompt των Windows. Παρακάτω, δίνουμε ένα ενδεικτικό παράδειγμα εκτέλεσης του script περιλαμβάνοντας τα απαραίτητα ορίσματα που πρέπει να δοθούν ως είσοδοι από τον χρήστη.

```
python rop_script.py shellcode.txt 0x00459109 10 2 On
```

Τα ορίσματα έχουν την εξής σημασία:

- Πρώτο όρισμα (π.χ. shellcode.txt): Ο χρήστης θα πρέπει να δώσει ως πρώτο όρισμα, το όνομα του αρχείου στο οποίο περιέχεται το shellcode, που επιθυμεί να μετατρέψει σε μορφή ROP.
- Δεύτερο όρισμα (π.χ. 0x00459109): Θα πρέπει να δοθεί ως όρισμα η διεύθυνση μνήμης (τμήμα .text), στην οποία πρόκειται να αποθηκευτεί το πρώτο gadget μετά την παραγωγή του εκτελέσιμου. Για την εύρεση αυτής της διεύθυνσης είναι απαραίτητη η χρήση κάποιου debugger. Με βάση αυτή την διεύθυνση, γίνεται μετέπειτα ο υπολογισμός των διευθύνσεων των υπόλοιπων gadget.
- Τρίτο όρισμα (π.χ. 10): Θα πρέπει να δοθεί ως είσοδος από τον χρήστη, ο αριθμός των εντολών assembly του shellcode που επιθυμεί να ενσωματωθούν σε gadgets.
- Τέταρτο όρισμα (π.χ. 2): Αναφέρεται στο αριθμό των εντολών assembly του αρχικού shellcode που θα περιλαμβάνει κάθε δημιουργηθέν gadget.
- Πέμπτο όρισμα (On/Off): Ενεργοποιείται η αντικατάσταση εντολών assembly του shellcode με ισοδύναμες όπου αυτό είναι δυνατό.

5.2 Έξοδος του script και δομή παραγόμενου αρχείου

Μετά την ολοκλήρωση της λειτουργίας του script, έχει δημιουργηθεί ένα αρχείο C (result.c), το οποίο περιέχει το shellcode σε μορφή ROP. Για να παραχθεί το μολυσμένο εκτελέσιμο, θα πρέπει να μεταγλωττιστεί ο κώδικας που περιέχεται στο συγκεκριμένο αρχείο. Ουσιαστικά, ο κώδικας C που παράγεται είναι ο ίδιος κάθε φορά, και το μόνο στοιχείο που διαφοροποιείται, είναι οι εντολές assembly που περιέχονται στο τμήμα της εντολής asm. Παρακάτω, δίνουμε ένα ενδεικτικό τμήμα του κώδικα C που έχει παραχθεί από το script.

```
void getprocaddr(){
        __asm {
                p459109:cld
                push 0x459110
                ret
                p45910a:call p459191
                p45910f:pushad
                push 0x45911c
                ret
                Next gadgets
        };
}
int main(int argc, char * argv[]){
                int cpt = 0;
                int i = 0;
                for(i =0; i < MAX_OP; i ++)  {
                        cpt++;
                }
                if(cpt == MAX_OP) {
                        char buflarge[] = { 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x09, 0x91, 0x45, 0x00};
                         test(buflarge);
                }
        }
void test(const char* buflarge) {
                char buf[5];
                memcpy(buf, buflarge, 16);
                printf(buf);
        }
```

Παρατηρούμε ότι, εισάγεται στην στοίβα του προγράμματος μέσω της υπερχείλισης του buffer, η διεύθυνση του πρώτου gadget. Με αυτόν τον τρόπο γίνεται η κλήση του πρώτου gadget και κατά συνέπεια, των εντολών assembly που περιέχονται στην εντολή asm. Κάθε gadget τελειώνει με τις εντολές push και ret. Με την εντολή push εισάγεται στην στοίβα του προγράμματος η διεύθυνση του επόμενου gadget και με την εντολή ret, γίνεται η κλήση του.

5.3 Disassembly του shellcode

Για την παραγωγή των εντολών assembly από τα αντίστοιχα machine codes του shellcode, δηλαδή για την επίτευξη της διαδικασίας disassembly, χρησιμοποιήθηκε το Capstone

Framework. Υποστηρίζει πολλές διαφορετικές αρχιτεκτονικές όπως είναι οι Arm, Arm64 (Armv8), M68K, Mips, PowerPC και Sparc, και έχει υλοποιηθεί με βάση την γλώσσα προγραμματισμού C.

Δίνουμε ως είσοδο στο Capstone Framework, την διεύθυνση στην οποία εντοπίζεται η πρώτη εντολή του shellcode μέσα στο δημιουργηθέν εκτελέσιμο (τομέας .text) καθώς και το ίδιο το shellcode. Ουσιαστικά, η επιθυμητή διεύθυνση δίνεται ως είσοδος από τον ίδιο τον χρήστη καθώς πρόκειται για το δεύτερο όρισμα εισόδου του rop_script.py. Αφού ολοκληρωθεί η εκτέλεση της λειτουργίας του Capstone Framework έχει δημιουργηθεί μία λίστα που περιέχει μικρότερες εμφωλευμένες λίστες. Κάθε εμφωλευμένη λίστα αποτελείται από την διεύθυνση της εκάστοτε εντολής assembly καθώς και από την ίδια την εντολή assembly. Οι μετέπειτα διαδικασίες του rop_script.py, όπως είναι η μετατροπή των εντολών σε ROP μορφή και η αντικατάσταση εντολών assembly με ισοδύναμες εντολές, γίνονται με βάση τα στοιχεία που περιέχει η συγκεκριμένη λίστα. Κατά αυτόν τον τρόπο, παράγονται οι εντολές assembly που ενσωματώνονται στην συνέχεια, στο μπλοκ της εντολής asm.

5.4 Αντικατάσταση εντολών assembly με ισοδύναμες εντολές

Μετά την ολοκλήρωση της διαδικασίας disassembly του shellcode, προχωράμε στην αντικατάσταση εντολών assembly με άλλες ισοδύναμες εντολές. Για να το επιτύχουμε αυτό, δημιουργούμε ένα λεξικό (dictionary). Στην γλώσσα προγραμματισμού Python, το λεξικό είναι μία δομή, της οποίας οι εγγραφές αποτελούνται από δύο στοιχεία: ένα κλειδί και τα δεδομένα που αντιστοιχούν στο συγκεκριμένο κλειδί. Κάθε εγγραφή του λεξικού που δημιουργήσαμε έχει την εξής δομή:

```
db = {‘εντολή προς αντικατάσταση’: [΄ισοδύναμη νέα εντολή’, διαφορά στον αριθμό των machine codes μεταξύ των δύο εντολών]}
```

Στην πραγματικότητα, κάθε κλειδί του λεξικού δείχνει σε μία λίστα δύο στοιχείων. Η λίστα αυτή περιέχει την ισοδύναμη νέα εντολή που πρόκειται να αντικαταστήσει κάποια από τις εντολές του shellcode καθώς και τον αριθμό των επιπρόσθετων bytes που θα δεσμευτούν στην μνήμη λόγω αυτής της αντικατάστασης. Είναι απαραίτητο να γνωρίζουμε τις όποιες πιθανές μετατοπίσεις στην μνήμη, έτσι ώστε στην συνέχεια, να μπορούμε να υπολογίσουμε τις διευθύνσεις των gadgets κατά την διαδικασία μετατροπής του shellcode σε μορφή ROP. Έτσι, για κάθε εντολή του shellcode, προσπελαύνουμε τα κλειδιά του λεξικού, με στόχο να βρεθεί μια πιθανή αντιστοίχιση μεταξύ της εντολής και του κλειδιού. Στην περίπτωση, που υπάρξει ισοδυναμία μεταξύ αυτών των δύο τότε προχωράμε στην αντικατάσταση της εντολής.

5.5 Μετατροπή του shellcode σε μορφή ROP

Για την δημιουργία των gadgets, ενσωματώνουμε στο τέλος κάθε εντολής assembly, τις εντολές push και ret. Η εντολή push περιέχει ως όρισμα την διεύθυνση στην οποία εντοπίζεται το επόμενο gadget της αλυσίδας ενώ με την εντολή επιστροφής ret γίνεται η κλήση του gadget αυτού. Παρακάτω, φαίνεται η αλληλουχία εντολών σε Python οι οποίες αποθηκεύουν στο παραγόμενο αρχείο του script (result.c), τις εντολές push και ret.

```
lengthOfAddition = counter * 6 + lengthOfInstruction + assemblyAdditionalLength
```

```
print("push 0x%x" %(thiselement+lengthOfAddition), file = fo)
print("ret", file = fo)
```

Για να υπολογισθεί η διεύθυνση του επόμενου gadget πρέπει να ληφθούν υπόψη τα εξής:

- counter: Αποθηκεύει τον αριθμό των gadgets που έχουν δημιουργηθεί έως αυτό το σημείο. Ο αριθμός αυτός πολλαπλασιάζεται επί έξι για να βρεθεί η πρόσθετη απόκλιση εξαιτίας της προηγούμενης εισαγωγής εντολών push και ret. Να σημειωθεί ότι οι εντολές push και ret, καταλαμβάνουν κάθε φορά 6 bytes μνήμης.
- lengthOfInstruction: Στην μεταβλητή αυτή αποθηκεύεται το μήκος της τρέχουσας εντολής assembly σε bytes.
- assemblyAdditionalLength: Η μεταβλητή αυτή, αποθηκεύει τον αριθμό των επιπρόσθετων bytes, λόγω της αντικατάστασης εντολών assembly με άλλες ισοδύναμες. Όπως αναφέρθηκε στην προηγούμενη ενότητα, στο λεξικό που είναι υπεύθυνο για τις αντικαταστάσεις εντολών, διατηρούμε τις διαφορές στον αριθμό των bytes όσον αφορά την νέα εντολή και την εντολή που αντικαταστάθηκε.

Αν προσθέσουμε όλες τις παραπάνω μεταβλητές στην διεύθυνση της τρέχουσας εντολής assembly (thiselement), τότε έχουμε υπολογίσει την διεύθυνση του αμέσως επόμενου gadget. Με αυτόν τον σχετικά απλό τρόπο έχουμε καταφέρει να δημιουργήσουμε μια αλυσίδα από ROP gadgets. Τα gadgets που δημιουργήθηκαν εγγράφονται στο τμήμα της εντολής asm, του αρχείου result.c, το οποίο αρχείο αποτελεί και την τελική έξοδο του rop_script.py.

5.6 Αποτελέσματα

Για αξιολογήσουμε την αποτελεσματικότητα του rop_script.py χρησιμοποιήσαμε την ιστοσελίδα VirusTotal. Μετατρέψαμε σε μορφή ROP το Reverse TCP Shell, το οποίο παρέχεται από το Metasploit Framework και στην συνέχεια, παρήγαμε το αντίστοιχο εκτελέσιμο αρχείο μεταγλωττίζοντας τον κώδικα C που περιλαμβάνονταν στο αρχείο result.c (έξοδος του rop_script.py). Για την μεταγλώττιση του κώδικα C, χρησιμοποιήθηκε το Microsoft Visual Studio.

Εκτελέσαμε το rop_script.py για διαφορετικές εισόδους και παρήγαμε τα αντίστοιχα εκτελέσιμα αρχεία. Για παράδειγμα:

- Μετατροπή σε μορφή ROP των τριάντα πρώτων εντολών assembly του shellcode.
- Δημιουργία δεκαπέντε gadgets, τα οποία θα αποτελούνται από δύο εντολές assembly του shellcode το καθένα.
- Εκτέλεση των παραπάνω παραλλαγών, έχοντας ενεργοποιημένη την δυνατότητα αντικατάστασης εντολών με άλλες ισοδύναμες εντολές.

Σαρώσαμε τα παραγόμενα εκτελέσιμα, χρησιμοποιώντας την ιστοσελίδα VirusTotal, με το ποσοστό αποφυγής των antivirus να ανέρχεται στο 96,4% για όλες τις διαφορετικές περιπτώσεις.

6. Συμπεράσματα

Τα περισσότερα antivirus βασίζονται σε υπογραφές και ήπιες τεχνικές δυναμικής ανάλυσης, όπως είναι ο καθορισμός της συμπεριφοράς του εκάστοτε προγράμματος, για την εξακρίβωση εάν πρόκειται για κάποιο είδος κακόβουλου κώδικα. Έτσι, κωδικοποιώντας τον κακόβουλο κώδικα στο ROP ισοδύναμό του και πραγματοποιώντας στοιχειώδεις αντικαταστάσεις εντολών, μπορούν να ξεπεραστούν οι μηχανισμοί των antivirus που βασίζονται στην ταυτοποίηση ήδη υπαρχουσών υπογραφών. Όσον αφορά, την δυναμική ανάλυση που διεξάγεται από τα antivirus, μπορεί να ξεπεραστεί εισάγοντας και εκτελώντας μη κακόβουλες εντολές, πριν την εκτέλεση του επιθυμητού κώδικα, καθώς τα antivirus σαρώνουν το εξεταζόμενο πρόγραμμα για περιορισμένο χρονικό διάστημα.

Οι τεχνικές που παρουσιάστηκαν στα πλαίσια αυτής της εργασίας θα μπορούσαν να ανιχνευθούν από τα antivirus, αν για παράδειγμα, δημιουργούταν υπογραφές για συγκεκριμένες δομές οι οποίες αποτελούνται από εντολές σε μορφή ROP. Ωστόσο, ακόμα και μία τέτοια λύση δεν φαντάζει απόλυτα αξιόπιστη καθώς μικρές διαφοροποιήσεις στον τρόπο μετατροπής των εντολών σε ROP, θα μπορούσαν να καταστήσουν τα antivirus αναποτελεσματικά. Επίσης, μία αξιόπιστη λύση δεν είναι εύκολη στον σχεδιασμό της και στην συγκεκριμένη περίπτωση, οι ρεαλιστικές πιθανότητες υλοποίησής της φαίνεται να συγκεντρώνουν μικρό ποσοστό.

6.1 Πιθανές μελλοντικές προεκτάσεις

Στο rop_script.py, το οποίο δημιουργήθηκε για την αυτοματοποίηση της παραγωγής ενός εκτελέσιμου που φέρει το επιθυμητό shellcode σε μορφή ROP, θα μπορούσαν να προστεθούν ορισμένες διαδικασίες και αλγόριθμοι για την πιο έξυπνη εκτέλεση μερικών από των λειτουργιών του. Μπορούν να εντοπιστούν δύο κύρια χαρακτηριστικά του συγκεκριμένου προγράμματος που θα μπορούσαν να αναβαθμιστούν και επομένως να εκτελούνται όσο το δυνατόν αποτελεσματικότερα.

Ειδικότερα, ο τρόπος που γίνεται η αντικατάσταση εντολών του shellcode με ισοδύναμες εντολές, θα μπορούσε να χαρακτηριστεί ως στατικός καθώς για την διεκπεραίωση της διαδικασίας αυτής, γίνεται χρήση ενός λεξικού (dictionary). Έτσι, για την αντικατάσταση μιας εντολής γίνεται αναζήτηση στο λεξικό, το οποίο περιέχει πιθανές ισοδυναμίες και στην περίπτωση που η εντολή του shellcode περιέχεται στο συγκεκριμένο λεξικό, τότε ακολουθεί η αντικατάστασή της. Η βέλτιστη λύση θα ήταν η δημιουργία ενός μεταγλωττιστή, ο οποίος θα δημιουργούσε με δυναμικό τρόπο τις κατάλληλες ισοδυναμίες, αποθηκεύοντας για κάθε εντολή την κατάσταση στην οποία βρίσκεται κάθε καταχωρητής. Διεξάγοντας την κατάλληλη ανάλυση κάθε εντολής θα μπορούσε να εντοπιστεί το είδος της εκάστοτε εντολής και εν συνεχεία, να γίνεται αντικατάστασή της με άλλες ισοδύναμες εντολές, οι οποίες θα χρησιμοποιούν τους μη δεσμευμένους καταχωρητές.

Ακόμα, μία σημαντική παρατήρηση είναι το γεγονός ότι, ορισμένες από τις εντολές assembly που παράγονται μετά το τέλος της διαδικασίας disassembly του shellcode, από το Capstone Framework, ενδέχεται να μην αναγνωρίζονται ως έγκυρες από τον inline assembler του Microsoft Visual Studio. Ως εκ τούτου, απαιτείται να βρεθούν οι εντολές

αυτές και να γίνεται η αντικατάστασή τους από τις αντίστοιχες εντολές, οι οποίες αναγνωρίζονται από το Microsoft Visual Studio. Σε διαφορετική περίπτωση, ενδέχεται να μην είναι δυνατή η μεταγλώττιση του κώδικα C που παράγεται από το rop_script.py εξαιτίας αναντιστοιχιών που αφορούν τις εντολές assembly.

7. Βιβλιογραφία

[1] Shellter. Διαθέσιμο στο: https://www.shellterproject.com [Πρόσβαση 27 Ιανουαρίου 2017]

[2] Injecting Shellcode into a Portable Executable (PE) using Python. Διαθέσιμο στο: http://www.debasish.in/2013/06/injecting-shellcode-into-portable.html [Πρόσβαση 27 Ιανουαρίου 2017]

[3] ROPInjector. Διαθέσιμο στο: https://github.com/gpoulios/ROPInjector [Πρόσβαση 27 Ιανουαρίου 2017]

[4] Mohan V., Hamlen K. W., 2012. Frankenstein: Stitching Malware from Benign Binaries, *WOOT'12 Proceedings of the 6th USENIX conference on Offensive Technologies*, pp. 77-84

[5] Shacham H., 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), *Proceedings of the 14th ACM conference on Computer and communications security, ACM Press*, pp. 552-61

[6] Erickson J., 2008. Hacking: The art of exploitation, *No Starch Press*

[7] Anley C., Heasman J., Linder F., Richarte G., 2007. The Shellcoder's Handbook: Discovering and Exploiting Security Holes, *Wiley Publishing*

[8] Koret J., Bachaalany E., 2015. The Antivirus Hacker's Handbook, *Wiley Publishing*

[9] "Allwin URLDownloadToFile + WinExec + ExitProcess" shellcode. Διαθέσιμο στο: https://www.exploit-db.com/exploits/24318/ [Πρόσβαση 27 Ιανουαρίου 2017]

[10] Metasloit Framework. Διαθέσιμο στο: https://www.metasploit.com [Πρόσβαση 27 Ιανουαρίου 2017]

[11] VirusTotal. Διαθέσιμο στο: https://www.virustotal.com [Πρόσβαση 27 Ιανουαρίου 2017]

[12] Online x86 / x64 Assembler and Disassembler. Διαθέσιμο στο: https://defuse.ca/online-x86-assembler.htm [Πρόσβαση 27 Ιανουαρίου 2017]