Κεφάλαιο 4
Εφαρμογή της μεθόδου εντοπισμού Buffer Overflow Exploit «Abstract Execution of Payload» και αναζήτηση των Return Address για Validation

4.1 Γενικά

Στα πρώτα κεφάλαια αυτής της εργασίας παρουσιάσθηκε ο τρόπος που γίνεται μία επίθεση βασισμένη στην **εκμετάλλευση(Exploit)** της **αδυναμίας υπερχείλισης μνήμης(buffer overflow vulnerability)** μιας εφαρμογής ή υπηρεσίας. Ακόμα παρουσιάστηκαν αναφορικά διάφοροι τρόποι αντιμετώπισης του προβλήματος είτε σε **επίπεδο δικτύου** είτε σε **επίπεδο εφαρμογής** είτε σε **επίπεδο συστήματος**. Καμία από αυτές τις λύσεις δεν θεωρείται πανάκια είτε επειδή είναι ασύμφορη οικονομικά(βλέπε compilation με boundary checking) είτε γιατί δεν μπορεί να αντιμετωπίσει όλες τις περιπτώσεις του προβλήματος (βλέπε NIDS).

Σε αυτό το κεφάλαιο θα παρουσιαστεί ένας τρόπος αντιμετώπισης του προβλήματος σε επίπεδο δικτύου που βασίζεται στην μέθοδο “**A**bstract **E**xecution of **P**ayload”(**A.E.P.**) και αναφέρεται στο προηγούμενο κεφάλαιο. Ακόμα σε αυτή την λύση θα προστεθεί και ένας μηχανισμός που θα ψάχνει για πιθανές ακολουθίες από **R**eturn **A**ddresses. Το τελευταίο θα είναι ουσιαστικά μία «**επιβεβαίωση**» ότι αυτό που βρήκε ο **A.E.P** είναι Buffer Overflow ενώ στην περίπτωση που δεν θα βρεθούν **R.A** τότε απλά θα **αναφέρεται ότι υπάρχει μια πιθανότητα το πακέτο κατά «σύμπτωση»** να θεωρήθηκε βλαβερό(malicious) ενώ δεν ήταν.

Η **λύση** αυτή ουσιαστικά **συνδυάζει τις παραπάνω μεθόδους με το IDS**. Έτσι ο αλγόριθμος αυτός δεν θα έχει σαν ρόλο πλέον να εμποδίζει την εκτέλεση ένος Buffer Overflow exploit αλλά κυρίως να το εντοπίζει καθώς αυτό κινείται μέσα στο δίκτυο. Αυτός **ο μηχανισμός θα είναι μία ισχυρή ένδειξη** για τον διαχειριστή του δικτύου **ότι το δίκτυο** του **δέχεται επίθεση,** βασισμένη σε **αδυναμίες υπερχείλισης μνήμης,** καθιστώντας τον σε θέση να αντιδράσει. Τέλος σε μία ιδανική περίπτωση που τα **False Positive** είναι ελάχιστα θα μπορεί το IDS να παίζει το ρόλο του IRS(Intrusion Response System) και να καταστρέφει το πακέτο πριν αυτό φτάσει στον προορισμό του. Με αυτό τον τρόπο θα γίνεται πιο εύκολο να εφαρμοστεί ο παραπάνω αλγόριθμος σε ένα ολόκληρο δίκτυο παρά να γίνει module σε κάθε μια από τις εφαρμογές που χρησιμοποιούνται στο δίκτυο που είναι αρκετά επίπονη διαδικασία και με μεγάλο οικονομικό κόστος.

Αρχικά θα γίνει μια αναλυτική περιγραφή που βασίζεται η ιδέα του **A.E.P** και γιατί ο *Thomas Toth* και *Christopher kruegel* χαρακτηρίζουν τον αλγόριθμο τους σαν Accurate(ακριβείας). Στην συνέχεια θα περιγραφεί ποιος είναι αυτός ο αλγόριθμος, πώς λειτουργεί και ποιοι **είναι οι σταθεροί παράγοντες που τον επηρεάζουν**. Κατόπιν θα περιγραφεί ο λόγος που πρόσθεσα την αναζήτηση του **RA**. Τέλος περιγράφεται η διαφορά και οι δυσκολίες της υλοποίησης του αλγορίθμου σαν **module ενός IDS** αντί σαν module σε κάθε υπηρεσία, όπως για τον **Apache Sever**.

4.2 Παρατηρώντας τα Buffer Overflow Exploits με σκοπό την ανάπτυξη ενός αλγορίθμου εντοπισμού

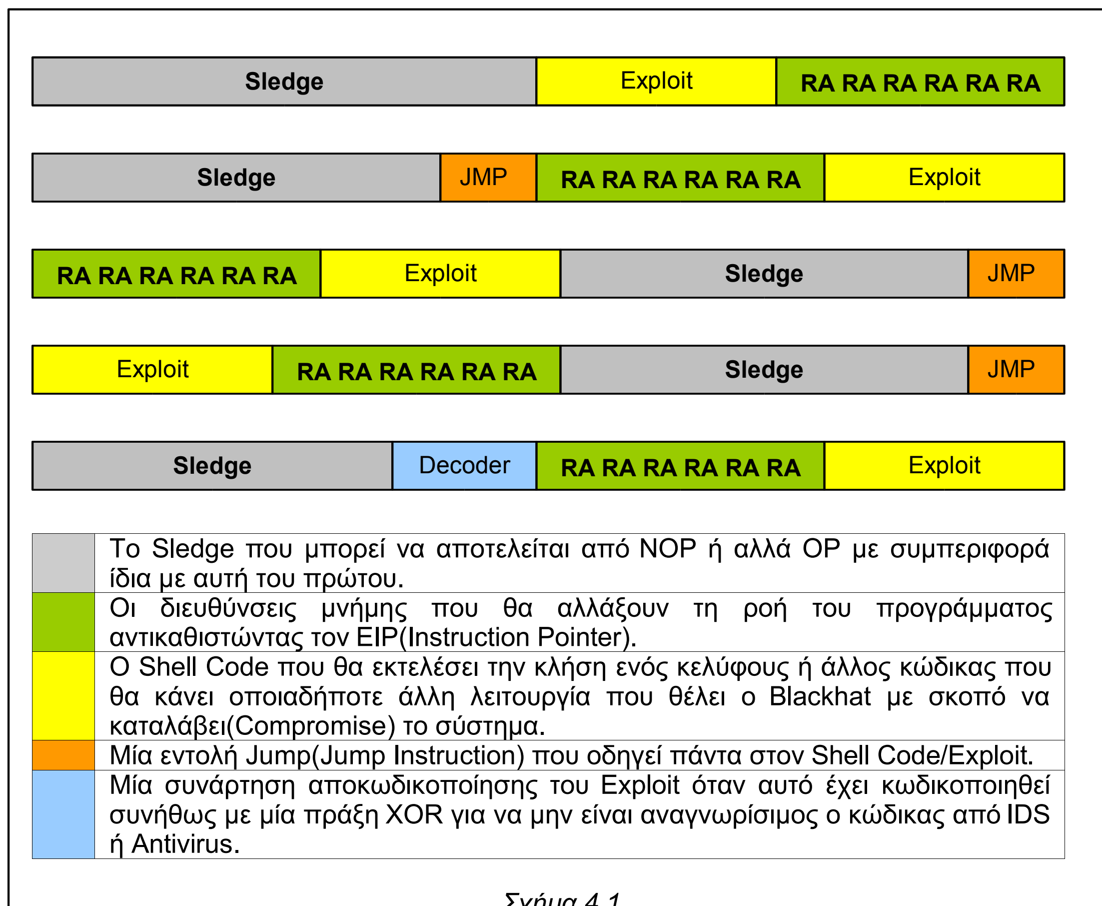
4.2.1 Γενικά

Όπως περιγράφονται στο **Κεφάλαιο 2** τα **Buffer Overflow Exploit** εξαρτώνται από διάφορους παράγοντες όπως είναι το **λειτουργικό σύστημα** και η **τεχνολογία του επεξεργαστή** δηλαδή η **τεχνολογία της πλατφόρμας** οπου η εφαρμογή αυτή εκτελείται. Ακόμα εξαρτάται από την **γλώσσα προγραμματισμού** που υλοποιήθηκε η εφαρμογή δηλαδή αν αυτή είναι μεσαίου ή χαμηλού επιπέδου. Τέτοιες γλώσσες όπως η C/C++ που *δεν κάνουν έλεγχο ορίων (boundary checking)* στα buffer τους και γενικότερα στις δομές δεδομένων τους. Τέλος εξαρτάται από τις **βιβλιοθήκες** που χρησιμοποιούν όπως τα **DLL** ή άλλες που από αυτές **κληρονομούν ένα σύνολο από αδυναμίες**. Γενικότερα η δημιουργία

ενός τέτοιου **Exploit** είναι ένα πολυσύνθετο πρόβλημα και εξαρτάτε από πολλούς διαφορετικούς παράγοντες που το καθιστά **κατά κάποιο τρόπο «μοναδικό».**

Το πρώτο πράγμα που θα πρέπει να γίνει για να βρεθεί ένας τρόπος να εντοπιστεί μία τέτοια επίθεση είναι να παρατηρηθούν διάφορες περιπτώσεις **Buffer Overflow Exploits** ώστε να μπορέσουν τα εξαχθούν τα κοινά τους χαρακτηριστικά. Τα χαρακτηριστικά αυτά θα είναι και οι παράγοντες που θα ελέγχονται κάθε φορά για να διαπιστωθεί αν ένα **IP πακέτο ή άλλος φορέας περιέχει είναι Buffer Overflow Exploit(B.O.E).** Άλλος φορέας μπορεί να είναι για παράδειγμα κάποιο αρχείο. Επειδή όμως η εφαρμογή θα γίνει σε επίπεδο δικτύου που είναι ο πλέον συνηθισμένος τρόπος μεταφοράς του **B.O.E** από τον θύτη στο θύμα θα αναφέρεται μόνο το πακέτο σαν φορέας για λόγους συντομίας, χωρίς φυσικά να αποκλείονται με αυτό τον τρόπο και οι άλλοι φορείς.

Από όλα τα προηγούμενα κεφάλαια διαπιστώνουμε ότι ένα Buffer Overflow exploit δεν είναι παρά ένα πολύ καλά **διαμορφωμένο (Crafted) String δηλαδή μια ακολουθία από Bytes.** Αυτό το **String** όταν θα δοθεί σαν είσοδος στο *αδύναμο(Vulnerable)* πρόγραμμα θα του προκαλέσει υπερχείλιση μνήμης (Buffer Overflow). Ο σκοπός της υπερχείλισης είναι να αναγκάσει το σύστημα αντί να συνεχίσει να εκτελεί το πρόγραμμα κανονικά, να εκτελέσει το ίδιο το String που θα έχει αλλάξει πλέον την σημασία του για τον υπολογιστή και από δεδομένα(Data) θα θεωρείται πλέον κώδικας(Code) δηλαδή *εκτελέσιμες εντολές(CPU instructions)* για τον επεξεργαστή. Παρατηρώντας την μορφή που έχει ένα τέτοιο Exploit String όπως αυτό παράγεται από το σχετικό πρόγραμμα που έχει φτιάξει ο **Blackhat Hacker**, βλέπουμε ότι υπάρχουν οι παρακάτω περιπτώσεις:



Σχήμα 4.1

Ενδεχομένως να υπάρχουν και πολλές άλλες περιπτώσεις. Η ποικιλία της γραφής του ίδιου Exploit με διαφορετική **«Γεωμετρία»** επιτρέπει στον Hacker να ξεγελάσει ένα NIDS ή Anti-Virus πρόγραμμα που κάνει Signature Based Detection. Αυτό θα το κάνει ο hacker χωρίς καν να αλλάξει τα κομμάτια του String άλλα μόνο τη σειρά που αυτά είναι γραμμένα. Το σημαντικό όμως που παρατηρεί κανείς είναι ότι τα Exploit έχουν κάποια **«Γεωμετρία»** με κοινά

χαρακτηριστικά μεταξύ τους, εκτός από ειδικές περιπτώσεις. Λεπτομέρειες για το πού χρησιμεύει αυτή η παρατήρηση θα παρουσιαστούν παρακάτω.

4.2.2 Παρατηρώντας το ShellCode ή Exploit

Συνεχίζοντας να παρατηρούμε πώς είναι ένα Exploit, το πρώτο πράγμα που θα σκεφτόταν κανείς θα ήταν να ελέγχει κάθε φορά το κύριο κορμό του B.O.E δηλαδή το ShellCode. Αυτό που κάνουν τα περισσότερα NIDS σήμερα είναι να ψάχνουν να βρουν κάποια χαρακτηριστικά μέσα στο πακέτο που ανήκουν σε ένα Exploit, και ειδικότερα στο Shell Code που **είναι απόδειξη** ότι το πακέτο **περιέχει B.O.E**. Τέτοια χαρακτηριστικά είναι για παράδειγμα τα String με **περιεχόμενο "/bin/sh" για το UNIX ή "CMD.EXE" για τα Ms Windows**. Όπως όμως περιγράφεται και στα προηγούμενα κεφάλαια υπάρχουν δυο τουλάχιστόν λόγοι για τους οποίους δεν είναι ιδιαιτέρα αποδοτικός αυτός ο τρόπος εντοπισμού.

Ο πρώτος είναι ότι ο κώδικας δεν είναι πάντα κώδικας που δίνει κέλυφος(Shell) αλλά που κάνει κάτι τελείως διαφορετικό. Για παράδειγμα στα Ms Windows επειδή είναι μεγάλος ο κώδικας του Shellcode, στις περιπτώσεις που το Buffer προς εκμετάλλευση είναι πολύ μικρό, χρησιμοποιείται **άλλος κώδικας δηλαδή άλλο Exploit**. Αυτό που συνήθως κάνει αυτό το Exploit είναι να κατεβάζει το shellCode από μία URL διεύθυνση στο Internet και στην συνέχεια το εκτελεί φέρνοντας έτσι ακριβώς τα ίδια αποτελέσματα χωρίς να χρειαστεί το BOE να είναι πολύ μεγάλο.

Ο δεύτερος τρόπος και ο πιο συνηθισμένος είναι να **κρυπτογραφείται ο Shellcode** και όταν αρχίσει να εκτελείται το exploit λίγο πριν εκτελεστεί το κομμάτι του ShellCode να **αποκρυπτογραφείται** με την κατάλληλη συνάρτηση.

Συμπεραίνοντας καταλήγουμε ότι η χρήση του Shellcode σαν **κριτήριο** κάνει τον εντοπισμό των B.O.E εξαιρετικά δύσκολο λόγω των **πρακτικά άπειρων παραλλαγών και περιπτώσεων** που μπορούν να αντικαταστήσουν το ShellCode (ή Exploit). Το αποτέλεσμα είναι ότι **εσκεμμένα ή μη** ο Blackhat καταφέρνει να κοροϊδέψει ένα **μηχανισμό εντοπισμού B.O.E** που χρησιμοποιεί σαν κριτηριο το Shellcode.

4.2.3 Παρατηρώντας το RA

Παρατηρούμε ότι είναι εξαιρετικά δύσκολο να εντοπίζονται τα buffer overflow από το ShellCode ενώ οι παραλλαγές του ίδιου Exploit αποκλείεται να εντοπιστούν **παρά μόνο αφού αυτό εξαπλωθεί και γίνει τελικά γνωστό**. Το επόμενο που θα σκεφτόταν κάνεις θα ήταν να εξετάσει τις RA. Αυτό που θα πρέπει λοιπόν να γίνει είναι να βρούμε μέσα σε ένα string διευθύνσεις μνήμης που ξέρουμε εξαρχής ότι είναι αυτές που κατά κανόνα χρησιμοποιούν οι hacker για να κάνουν την επίθεση ή γενικότερα **που ανήκουν σε διευθύνσεις μνήμης της περιοχή της στοίβας(Stack)**.

Η επιλογή να χρησιμοποιείται σαν κριτήριο οι R.A. είναι καλύτερη σε σχέση με το ShelCode γιατί ο BlackHat Hacker δεν μπορεί να κρυπτογραφήσει τις **διευθύνσεις επιστροφής(Return Address).** Βεβαιωνόμαστε ακόμα περισσότερο ότι η επιλογή της χρήσης των RA σαν κριτήριο είναι καλή γιατί παρατυρείται ότι συνήθως μία **R.A** υπάρχεί πολλές φορές μέσα σε ένα **B.O.E**. Όμως υπάρχουν διάφοροι παράγοντες που **δεν επιτρέπουν σε αυτή την μέθοδο να είναι το βασικό κριτήριο** για να διαπιστωθεί αν ένα πακέτο περιέχει **B.O.E.**

Ο βασικότερος παράγοντας που **αποθαρρύνει** αυτή τη μέθοδο από το να χρησιμοποιηθεί σαν ο κύριος τρόπος εντοπισμού είναι επειδή σε παρά πολλές περιπτώσεις **είναι αρκετή η ύπαρξη μόνο μιας RA μέσα σε ένα String ώστε αυτό να είναι πετυχημένο.** Επιπλέον μια διεύθυνση επιστροφής δεν είναι τίποτα άλλο παρά μόνο μια λέξη από 4 χαρακτήρες δηλαδή 4 byte. Άρα η πιθανότητα **έστω κατά «σύμπτωση» να βρεθούν έστω και μία φορά** τέτοιες τετράδες είναι **μεγάλη**.

Ένας δεύτερος παράγοντας που αποθαρρύνει την επιλογή αυτή είναι ότι σε πολλές περιπτώσεις ο Blackhat χρησιμοποιεί αφενός R.A. που δεν ανήκουν στην περιοχή της στοίβας, αφετέρου οι διευθύνσεις αυτές δεν είναι προσβάσιμες από όλες τις εφαρμογές. Αυτό έχει σαν αποτέλεσμα να χρειάζεται να αναζητώνται περιοχές που δεν είναι εντός της στοίβας **ειδικά στα Exploit που εκμεταλλεύονται τα DLL.** Ακόμα και αν ελέγχονται όλες οι περιοχές

εκτός και εντός της στοίβας το πρόβλημα δεν λύνεται αφού αυτές οι περιοχές που **βρέθηκαν** μπορεί να μην είναι προσβάσιμες από την εφαρμογή. Στην περίπτωση αυτή το B.O.E δεν θα πετύχει γιατί το **λειτουργικό σύστημα** θα εντοπίσει το λάθος και θα σταματήσει την εκτέλεση της εφαρμογής. Τελικά διαπιστώνουμε ότι το πιθανότερο είναι να έχουμε πολλά **False positive** γιατί οι R.A που θα βρεθούν μπορεί να μην είναι αφενός ικανες για να πετύχει το BOE αφετέρου να μην είναι μέρος ενος παραγαμτικού B.O.E.

Τέλος ένας ακόμα παράγοντας είναι ότι η ακολουθεία των **Return Address** μπορεί να γραφτεί με διάφορες παραλλαγές όπως και το Sledge που θα εξετάσουμε παρακάτω. Οι παραλλαγές αυτές μπορεί να είναι οι εξής:

Sledge | Exploit | **RA**

RA RA RA RA RA RA RA RA RA RA RA RA RA RA RA RA RA RA RA **...**

RA (RA-2) (RA-4) (RA-8) (RA-12) (RA-16) RA (RA-2) (RA-4) (RA-8) (RA-12) **...**

RA (RA+2) (RA+4) (RA+8) (RA+12) (RA+16) RA (RA+2) (RA+4) (RA+8) **...**

Σχήμα 4.2

Όπως φαίνεται στα παραπάνω χαρακτηριστικά παραδείγματα μπορεί η **RA** να μη **επαναλαμβάνεται** πολλές φορές η ίδια αλλά και οι παραλλαγές της δηλαδή με **μικρό offset** από τη αρχική RA. Αυτό θα έχει την **ίδια ακριβώς επιτυχία ειδικά όταν υπάρχει πολύ μεγάλο Sledge.** Σε αυτή την περίπτωση το μόνο που χρειάζεται να κάνουμε είναι να βρίσκουμε κάθε φορά όχι μόνο μία **RA** πολλές φορές αλλά τη **RA** με τις παραλλαγές της.

Συμπεραίνουμε λοιπόν με βάση τα παραπάνω ότι η αναζήτηση για **Return Address** μέσα σε ένα String μπορεί να είναι **μία ένδειξη** ότι αυτό είναι B.O.E, αλλά δεν μπορεί να είναι το βασικό κριτήριο εντοπισμού. Όμως ένας τέτοιος μηχανισμός μπορεί να **είναι μια καλή μέθοδος επιβεβαίωσης(Validation)** ότι ένα String είναι ή περιέχει Buffer Overflow Exploit.

4.2.4 Παρατηρώντας το Sledge.

Όπως προαναφέρεται στα **Κεφάλαια 1 και 2** Sledge ονομάζουμε το κομμάτι του B.O.E. που αποτελείται από μια σχετικά μεγάλη ακολουθία από NOP. Το NOP είναι μια εντολή που όταν εκτελείται δεν κάνει καμία άλλη ενέργεια **παρά μόνο να χρησιμοποιεί έναν κύκλο ρολογιού της CPU**. Αυτή η εντολή χρησιμοποιείται για περιπτώσεις συγχρονισμού ενώ **για τον Backchat αυτό που κάνει είναι να οδηγεί τον IP(Instruction Pointer)** προς το πραγματικό Exploit(Shellcode) αυξάνοντας δραματικά την πιθανότητα επιτυχίας του B.O.E.

Η χρήση του NOP σαν κριτήριο για το εντοπισμό των B.O.E είναι μία πολύ καλή μέθοδος σε σχέση με τις δύο προηγούμενες μεθόδους. Οι παράγοντες που την μέθοδο αυτή την καθιστούν ιδιαίτερα αποτελεσματική είναι :

1. Πρέπει να είναι **πάντα σε εκτελέσιμη μορφή** για να μπορέσει να πετύχει το BOE. Αυτό σημαίνει ότι **δεν μπορεί να είναι κρυπτογραφημένο** (το sledge) όπως το **ShellCode**.
2. Όταν **η ύπαρξη του Sledge είναι απαραίτητη** για να πετύχει το Exploit τότε σίγουρα αυτό θα βρίσκεται μέσα στο string **σε μεγάλη ποσότητα**.

Το μεγάλο όμως πρόβλημα που καθιστά την αναζήτηση των NOP μη ικανή συνθήκη για να θεωρηθεί ότι ένα πακέτο οτι περιέχει BOE είναι ότι το NOP δεν είναι η μοναδική εντολή του επεξεργαστή που έχει αυτή την ιδιαίτερη συμπεριφορά. Μάλιστα ειδικά για την περίπτωση της IA32(intel 32) τεχνολογίας υπάρχει ένα σύνολο πάνω από 50 εντολές που είναι Operational (OP) δηλαδή **δεν είναι η No Operation(NOP) αλλά μπορούν να παίξουν ακριβώς τον ρόλο του NOP**. Ενδεικτικά στον παρακάτω πίνακα φαίνεται ένα μικρό δείγμα του συνόλου των εντολών που περιγράφεται αναλυτικά στο *κεφάλαιο 2*.

Εντολή σε Assembly	Λειτουργία	OP code
AAA	ASCII Adjust After Addition	0x37
AAS	ASCII Adjust After Subtraction	0x3f
CWDE	Convert Word To Doubleword	0x98
CLC	Clear Carry Flag	0xf8
CLD	Clear Direction Flag	0xfc
CLI	Clear Interrupt	0xfa
CMC	Complement Carry Flag	0xf5

Όπως παρατηρεί κανείς κάθε μια από αυτές της εντολές δεν θα κάνει τίποτα παραπάνω από το να αναγκάσει την CPU να την εκτελέσει χωρίς καμία άλλη συνέπεια **για το B.O.E**. Το μονο που θα κάνει έιναι από το να μετακινήσει τον Instruction Pointer κατά ένα Byte. Έτσι κάθε Blackhat hacker θα μπορούσε να χρησιμοποιήσει κάθε μια από αυτές τις εντολές για να αντικαταστήσει την NOP και να **σχηματίσει ένα νέο Sledge**. Στην περίπτωση αυτή θα έλεγε κανείς ότι το μόνο που πρέπει να γίνει είναι να εντοπίζονται εκτός από ακολουθίες NOP και εντολές τέτοιου τύπου και ο συνδυασμός τους που πρακτικά θα ήταν ένας πεπερασμένος αριθμός.

Το πρόβλημα όμως μεγαλώνει από τον τρόπος που λειτουργεί η στοίβα των σύγχρονων 32bit υπολογιστών η οποία κάθε φορά αποθηκεύει(PUSH) και ανασύρει(POP) ομάδες των 4 byte άσχετα άν τα byte που χριάζεται να αποθυκευτούν διαιρούνται ακρυβώς με το 4. Αυτό καθιστά δυνατό να χρησιμοποιηθούν και εντολές πέρα του 1 byte δηλαδή και εντολές που δέχονται παραμέτρους. Αυτό προκαλεί δύο επιπλέον προβλήματα :

1. Το σύνολο των OP που μπορεί να αντικαταστήσει το NOP είναι πρακτικά **όλο το Instruction Set ενός επεξεργαστή**.
2. Κάθε εντολή μπορεί να έχει οποιαδήποτε παράμετρο ανάλογα με το εκάστοτε περιβάλλον που θα χρησιμοποιηθεί ώστε να πετύχει το Exploit.

Ο παρακάτω πίνακας δείχνει μερικά παραδείγματα Instruction που έχουν παραμέτρους και μπορούν να αντικαταστήσουν το NOP αρκεί φυσικά ο Blackhat να εξασφαλίσει ότι η αρχή της εκτελέσιμης εντολής είναι **μέσα στα όρια των 4byte που γίνονται push και pop κάθε φορά στην στοίβα**.

CPU Instruction γραμμένη σε Assemlby	Bytecode
Adc $0x70 %cl	80 d1 70
Adc $0x70d18070 %ecx	81 d1 70 80 d1 70
And $0x551220125 %eax	25 01 12 44
Jmprel 0x37	0xeb 0x37

ΠΑΡΑΤΗΡΗΣΗ
Οι παράγοντες που περιγράφονται παραπάνω **αυξάνουν δραματικά το σύνολο των εντολών αλλά και τον συνδυασμό τους** που μπορούν να αντικαταστήσουν το NOP και να σχηματίσουν ένα Sledge.

Θα συμπέραινε λοιπόν κανείς ότι δεν υπάρχει κανένα χαρακτηριστικό αρκετά σταθερό για να μπορέσει να βασιστεί κανείς ώστε να εντοπίσει τα Buffer Overflow Exploits. Το μόνο που είναι εύκολο και μέχρι στιγμής γίνεται είναι να **εντοπίζονται τα Buffer Overflow Exploits με Misused Based Detection** παρακολουθώντας ένα από τα παραπάνω χαρακτηριστικά ή πιάνοντας ειδικές περιπτώσεις των ήδη γνωστών B.O.E..

4.2.5 Οι <u>Συνέπειες της Συμπεριφοράς του Sledge</u> σαν παράγοντας για Anomaly Based Detection.

Παρατηρώντας όμως πιο προσεκτικά το Sledge **τότε αντιλαμβανόμαστε ότι πάντα είναι σε μεγάλη ποσότητα και πάντα είναι εκτελέσιμο**. Η κεντρική ιδέα λοιπόν είναι να βλέπουμε μέσα σε ένα String **ποια Set(σύνολα) από χαρακτήρες είναι εκτελέσιμα αλλά και σε ποιά ποσότητα υπάρχουν**. Δηλαδή με απλά λόγια θα έψαχνε κανείς ποιες από τις λέξεις που σχηματίζονται μέσα σε ένα String σημαίνουν NOP ή είναι NOP και αν αυτές είναι σε μεγάλη ποσότητα.

Η μέθοδος αυτή είναι μια **Anomaly Based Detection** μέθοδος γιατί με αυτή θα πρέπει να υπάρχει ένας αρκετά μεγάλος αριθμός από εκτελέσιμες εντολές μέσα σε ένα πακέτο για να θεωρηθεί ότι περιέχει ΒΟΕ. Αυτή η συμπεριφορά όπως θα δούμε παρακάτω δεν είναι συνηθισμένη και για αυτό αν παρατηρηθεί κάτι τέτοιο λέμε ότι παρατηρείται **μία ανωμαλία (Anomaly)** στα data αυτού του πακέτου.

Η επιλογή να χρησιμοποιηθεί η «**Εκτελεσιμότητα**» του sledge σαν κριτήριο επιλογής για το αν ένα String περιέχει B.O.E φαίνεται να **είναι αρκετά καλή**. Όμως και εδώ παρουσιάζεται ένα πρόβλημα. Το πρόβλημα είναι ότι **η πιθανότητα να βρεθούν κατά «σύμπτωση»** εκτελέσιμες εντολές **μέσα σε ένα string, ενώ αυτό δεν είναι ούτε εκτελέσιμο πρόγραμμα αλλά ούτε B.O.E, είναι μεγάλη**. Το πώς συμβαίνει αυτό περιγράφεται παρακάτω.

<u>4.2.5.1 Το πρόβλημα της «Σύμπτωσης»</u>

Στον ηλεκτρονικό υπολογιστή είναι γνωστό ότι τα πάντα εκφράζονται με αριθμούς. Ακόμα και οι δομές και τα σύμβολα που χρησιμοποιούνται για την αναπαράσταση της πληροφορίας στο Ηλεκτρονικό Υπολογιστή εκφράζονται και αυτά ως αριθμοί. Το ερώτημα λοιπόν είναι πώς γίνεται αυτό.

Η απάντηση στο παραπάνω είναι ότι οι κατασκευαστές του Η/Υ αποφάσισαν να δίνουν μια **συγκεκριμένη σημασία** ή αλλιώς εννοιολογικό περιεχόμενο σε κάθε αριθμό ή ομάδα αριθμών. Αναλόγως λοιπόν με την σημασία που έχει προκαθοριστεί για αυτή την ομάδα των αριθμών αυτοί θα περάσουν μέσα από τον κατάλληλο μηχανισμό αναπαραγωγής ή επεξεργαστή και θα παραχθεί η ανάλογη πληροφορία. Για παράδειγμα **περνώντας αυτή την πληροφορία**(σύνολο από αριθμούς) **από τον επεξεργαστή εικόνας** θα δούμε ένα ή παραπάνω χαρακτήρες να αποτυπώνονται στην οθόνη του υπολογιστή. Ενώ **αν το σύνολο αυτής της πληροφορίας βρεθεί σε κατάλληλη περιοχή στην μνήμη** του υπολογιστή τότε θα **θεωρηθεί σαν πρόγραμμα που η CPU θα προσπαθήσει να εκτελέσει**. Για να γίνουν όλα αυτά πιο ξεκάθαρα παρουσιάζονται μερικά πολύ απλά παραδείγματα:

Παράδειγμα 4.1

Όσοι είναι αρκετά εξοικειωμένοι με την C/C++ γνωρίζουν ότι όταν ο προγραμματιστής ορίσει ότι μία μεταβλητή είναι τύπου Char(χαρακτήρας) τότε ενώ σε άλλες γλώσσες προγραμματισμού δεν θα ήταν επιτρεπτό, στην C/C++ επιτρέπεται να καταχωρήσουμε και αριθμό.

```
Main(){
      Char c;
      c='7' ;      // θα κρατήσει τον χαρακτήρα 7
      c=55 ;       // θα κρατήσει τον χαρακτήρα 7
      c=0x37;      // θα κρατήσει τον χαρακτήρα 7

      printf("Content of c: %c",c); // θα τυπώσει τον χαρακτήρα 7
}
```

στις τρεις περιπτώσεις θα είναι ο χαρακτήρας 7 και όχι ο αριθμός 7. Αυτό θα συμβεί επειδή εμείς αρχικά ορίσαμε ότι η μεταβλητή **c** θα κρατά χαρακτήρες. Όταν λοιπόν δηλώνουμε μία μεταβλητή ως Char τότε στην πράξη αυτό που συμβαίνει είναι να καταχωρείται μέσα σε αυτή ένα byte δηλαδή ο αριθμός που εκφράζεται με 8bit. Αυτά τα 8 bit είναι γνωστό ότι είναι το μέγεθος του ενός χαρακτήρα στην C/C++. Ο αριθμός όμως αυτός επειδή είναι μέσα σε μία μεταβλητή που εμείς ορίσαμε ότι θα κρατά χαρακτήρες, θα πρέπει πριν εμφανιστεί στη οθόνη να αλλάξει σημασία και αντί για ένας απλός 8 bit αριθμός να γίνει ένας χαρακτήρας. Για να

γίνει αυτό ο αριθμός που η δεκαεξαδική του μορφή είναι **0x37** θα αντιστοιχιστεί στον πίνακα ASCII και από εκεί στην οθόνη θα τυπωθεί ο χαρακτήρας ‘7’.

Παρατηρώντας επίσης προσεκτικά την σύνταξη των παραπάνω εντολών διαπιστώνεται ότι όταν τον αριθμό τον πλαισιώνει το σύμβολο (‘) τότε το εννοιολογικό του περιεχόμενο δεν είναι πλέον αριθμός, αλλά χαρακτήρας. Άρα στο **c** δεν θα μπει το 7 αλλά ο αριθμός που στον ASCII πίνακα θα αντιστοιχεί στο 7.

Παράδειγμα 4.2
Όταν ορίστηκε στο προηγούμενο παράδειγμα η μεταβλητή **c** τότε στην **στοίβα(Stack)** και **συγκεκριμένα στο κομμάτι** αυτής που ανήκει στην συνάρτηση θα δεσμευτεί ένα Byte επειδή την **c** την διακηρύξαμε σαν char. Τότε λεμε ότι η περιοχή αυτή της στοίβας είναι η περιοχή που **κρατά τα δεδομένα** που χειρίζεται μία συνάρτηση. Όταν στο υπολογιστή θα ζητηθεί να χειριστεί τα δεδομένα που κρατά η μεταβλητή **c** τότε αυτός θα πάρει την κατάλληλη συνάρτηση από την **Code Area** της μνήμης και θα εκτελέσει επί της μεταβλητής ότι εντολές περιγράφονται στην περιοχή αυτή.

Αν παρατηρήσει κανείς τι υπάρχει στην Code Area θα διαπιστώσει ότι δεν είναι τίποτα άλλο από byte που πολλές φορές είναι αριθμοί που δεν διαφέρουν από τα δεδομένα. Εύλογο θα ήταν το ερώτημα πώς ο υπολογιστής ξεχωρίζει τα δεδομένα από τον εκτελέσιμο κώδικα. Η απάντηση είναι οτι, όπως είπαμε και σε προηγούμενα κεφάλαια, **από πριν έχει προδιαγραφεί** ποια θα είναι η περιοχή που τα byte θα θεωρούνται **εκτελέσιμος κώδικας (Executable Code)** και ποια θα είναι η περιοχή **δεδομένα(Data)**. Αν λοιπόν μπορούσαμε να βάλουμε την τιμή της μεταβλητής c στην Code Area τότε η CPU θα εκτελούσε την εντολή **“ASCII Adjust After Addition”**. Η εντολή σε Assembly γράφεται **AAA** ενώ η τιμή της είναι **0x37**.

> **ΠΑΡΑΤΗΡΗΣΗ**
> Παρατηρούμε λοιπόν ότι και στα δύο παραδείγματα ο χειρισμός (Handling) των αριθμών από τον υπολογιστή, δηλαδή τι θα κάνει τελικά με αυτούς, εξαρτάται από τις προδιαγραφές που έχουν δοθεί στους αριθμούς αυτούς.

4.2.5.2 Η λύση του προβλήματος της «σύμπτωσης»

Όπως είδαμε λοιπόν κατά **«σύμπτωση» η ίδια τιμή που αντιστοιχούσε** στον αριθμό 7 του πίνακα ASCII ήταν η ίδια τιμή που αντιστοιχεί και στη εντολή **AAA** της Assembly. Αυτή είναι η σύμπτωση που περιγράφεται παραπάνω. Δηλαδή το πρόβλημα του εντοπισμού ενός B.O.E. , με βάση τη «Εκτελεσιμότητα» του Sledge, είναι ότι υπάρχει μεγάλη πιθανότητα να βρεθούν μέσα σε ένα string **Op codes,** δηλαδή εκτελέσιμες εντολές, ενώ αυτές στην πραγματικότητα αντιστοιχούν σε απλά Data.

Το πρόβλημα άμεσα λύνεται αν παρατηρήσουμε στο sledge το εξής χαρακτηριστικό:
οι εκτελέσιμες εντολές πρέπει να είναι συνεχόμενες η μία μετά την άλλη χωρίς να παρεμβάλλονται χαρακτήρες που δεν είναι εκτελέσιμοι παρά μόνο οι Operands (παράγοντες) του opcode .

4.2.6 Συμπεράσματα (ικανές συνθήκες για BOE)

Παρατηρώντας τα B.O.E συμπεράναμε ότι οι συνθήκες για να χαρακτηριστεί ένα string στο payload ενός πακέτου σαν Buffer Overflow Exploit είναι :

1. Μέσα στο string να υπάρχουν Op Codes.
2. Επειδή **η πιθανότητα να υπάρχουν** κατά **«σύμπτωση»** Op Codes **είναι πολύ μεγάλη** πρέπει οι Op codes να **εμφανίζονται συνεχόμενα σαν μια αλυσίδα**.
3. Επειδή **η πιθανότητα να υπάρχουν** πολλές τέτοιες αλυσίδες **είναι μεγάλη** για αυτό πρέπει οι ποσότητα των συνεχόμενων εντολών Op code να είναι μεγάλη.

ΣΗΜΑΝΤΙΚΟ ΣΥΜΠΕΡΑΣΜΑ
Συμπεραίνουμε λοιπόν ότι ***το αποτέλεσμα που προκαλεί το Sledge*** μέσα σε ένα πακέτο, και γενικότερα σε ένα string, ***είναι μεγάλες αλυσίδες*** συνεχόμενων εκτελέσιμων εντολών(Op Codes). **Το μήκος των αλυσίδων θα είναι το κριτήριο διαχωρισμού** μεταξύ των πακετων με B.O.E και των πακέτων χωρίς B.O.E.

4.3 Abstract Execution of Payload ως αλγορίθμου εντοπισμού των B.O.E.

4.3.1 Γενικά

Για τον σχεδιασμό του αλγόριθμου πρέπει να οριστεί κάθε δομικό στοιχείο του String δηλαδή τι είναι και ποιό ρόλο θα παίζει για τον αλγόριθμο. Για αυτό το λόγο παρακάτω θα δοθούν ένα σύνολο από ορισμούς που θα χρησιμοποιηθούν για να σχεδιαστεί ο αλγόριθμος.

Ο όρος «Εκτελεσιμότητα»(Excitability) όπως προαναφέρθηκε είναι το βασικό χαρακτηριστικό που πρέπει να έχει κάθε λέξη(σύνολο από 1 και πάνω bytes) για να θεωρηθεί μέρος ενός Sledge. Για να είναι **μια λέξη εκτελέσιμη** θα πρέπει να **υπάκουει σε δύο συνθήκες** την **Correctness** και **Validity**(Εγκυρότητα).

- **Correctness:** Σαν σωστή(correct) ονομάζουμε μια ακολουθία από bytes(λέξη) όταν αυτή αντιστοιχεί σε κάποια εντολή του επεξεργαστή (Processor Instruction). Αυτό σημαίνει ότι την εντολή αυτή ο επεξεργαστής θα μπορεί να την αποκωδικοποιήσει και φυσικά να την εκτελέσει. Η ακολουθία αυτή πρέπει να αποτελείται από ένα σύνολο από bytes τόσα σε αριθμό όσα ακριβώς είναι αυτά που αντιστοιχούν στην εντολή του επεξεργαστή μαζί με τις παραμέτρους της(Operants). Σε όλες τις άλλες περιπτώσεις η ακολουθία θεωρείται λάθος.

- **Validity:** Μία ακολουθία από bytes(λέξη) ονομάζεται έγκυρη(valid) όταν είναι σωστή (correct) και επιπλέον όταν αυτή αναφέρεται σε περιοχές μνήμης(Memory Address) που είναι προσβάσιμες από την εκάστοτε διεργασία που θα εκτελέσει την ακολουθία. Σε περίπτωση που μια εντολή δεν έχει παράγοντες(operands) που αναφέρονται στην μνήμη του υπολογιστή τότε αυτομάτως θα θεωρείται ως valid.

  Σημείωση: Ο παράγοντας Valid έχει ιδιαίτερη σημασία, γιατί σε περίπτωση που από τους Opcodes ενός B.O.E υπάρχει αναφορά σε περιοχές μνήμης που δεν είναι προσβάσιμες από την εφαρμογή τότε το ίδιο το **λειτουργικό σύστημα** θα εντοπίσει το πρόβλημα σταματώντας την ροή του προγράμματος. Σε μερικές περιπτώσεις όμως, όπως στην εφαρμογή αυτής της πτυχιακής, όταν γίνεται προσπάθεια εντοπισμού B.O.E για όλες τις πιθανές εφαρμογές και όχι ειδικά για μια, ο έλεγχος αυτός δεν πρέπει να γίνεται. Γιατί δεν μπορεί να είναι γνωστό ποιες περιοχές μνήμης είναι προσβάσιμες και ποιες όχι για κάθε εφαρμογή. Λεπτομέρειες στη παράγραφο «Εντοπισμός της R.A σαν Validation Μηχανισμός».

Ο όρος «συνεχόμενες» που αναφέρθηκε παραπάνω σημαίνει ότι όλες οι μικρές εκτελέσιμες ακολουθίες θα πρέπει να είναι αλλεπάλληλες δηλαδή η μια μετά την άλλη σχηματίζοντας μια «αλυσίδα». Για αυτό το λόγο τα κομμάτια του String που θα έχουν τέτοιες «συνεχόμενες» εντολές θα λέμε ότι είναι **Instruction Chains(I.C.)** δηλαδή αλυσίδες εκτελέσιμων εντολών(Op Codes). Άρα ένα String που κομμάτια του έχουν την παραπάνω συμπεριφορά θα λέμε ότι περιέχει **I.C.** Ενώ την εκτελέσιμη ακολουθία ή λέξη λόγω της ιδιότητας της θα την ονομάζουμε και **Executable Instruction(E.I.)**

4.3.2 Προδιαγραφές αλγορίθμου Abstract Execution.

Για να μπορούν να εξετάζονται για κάθε πακέτο τα παραπάνω θα πρέπει ο αλγόριθμος που θα τα εξετάζει να έχει ένα σύνολο από προδιαγραφές:

1. Θα πρέπει να γνωρίζει ποιοι είναι οι OP codes του επεξεργαστή που θα εξετάσει.
2. Θα πρέπει να γνωρίζει τις συνέπειες της κάθε εντολής και αν χρειαστεί να ελέγχει αυτές τις συνέπειες.

3. Όλα αυτά πρέπει να γίνονται με έναν τρόπο που δεν θα χρειάζεται να εκτελεστεί πραγματικά το String ώστε να παρατηρηθούν οι συνέπειες.
4. Ο έλεγχος να γίνεται ταχύτερα από την πραγματική εκτέλεση του **I.C**.

Αυτές τις προδιαγραφές τις καλύπτει ο αλγόριθμος **Abstract Execution ή αλλιώς Αφαιρετικής Εκτέλεσης:**

ΟΡΙΣΜΟΣ του Abstract Execution
Abstract Execution ή αλλιώς Αφαιρετική Εκτέλεση είναι η διαδικασία κατά την οποία βρίσκουμε αν μια **ακολουθία από byte είναι εκτελέσιμη χωρίς να την δώσουμε να την εκτελέσει πραγματικά ο επεξεργαστής**, ενώ αν χρειαστεί να εξεταστούν οι συνέπειες της εκτέλεσης της αυτό γίνεται σε μορφή εξομοίωσης από τον ίδιο τον

ΟΡΙΣΜΟΣ
Ένα string ή **αλλιώς μία ακολουθία από bytes λέμε ότι είναι Abstact Executable** δηλαδή Αφαιρετικά Εκτελέσιμη **όταν αυτό θα περιέχει τουλάχιστο ένα I.C**.

Για να μπορεί να λειτουργήσει ο αλγόριθμος θα διαχωρίζουμε τις εντολές του **Instruction Set** σε δύο κατηγορίες σε αυτές που **αλλάζουν την κανονική ροή της εκτέλεσης** και σε **όλες τις άλλες.**

Η πρώτη κατηγορία περιλαμβάνει όλες τις εντολές όπως ***jmp, jne,* κτλ** δηλαδή όλες όσες χαρακτηρίζονται ως **jump instructions.** Το τέλος της **I.C**. καθορίζεται όταν αυτή καταλήγει σε ένα **JMP ή σε σκουπίδια(μη εκτελέσιμη εντολή).** Η jump instruction θα πρέπει να οδηγεί σε μία άλλη I.C. , σε μια άλλη jump instruction ή σε σκουπίδια για να καθοριστεί το τέλος της **«συνολικής» I.C.** Άρα το μήκος μιας **συνολικής IC** είναι το άθροισμα των μικρών I.C που την αποτελούν. Σε κάθε ακολουθία από bytes μπορούν να υπάρχουν πολλές I.C. που συνδέονται με jump instruction άρα και να σχηματίζουν πολλές διαφορικές **«συνολικές» IC** .

Η δεύτερη κατηγορία είναι οι υπόλοιπες εντολές του επεξεργαστή που στην πράξη δεν μας ενδιαφέρει το αποτέλεσμα θα έχουν όταν εκτελεστούν. Ο λόγος που **δεν μας ενδιαφέρει οφείλεται κυρίως στο ίδιο τον στόχο που έχει ο Blackhat hacker όταν αυτός χρησιμοποιεί το Sledge**. Δηλαδή, οι εντολές που βρίσκονται μέσα στο sledge έχουν σαν στόχο να οδηγήσουν τον Instruction Pointer προς τον ShellCode άρα δεν μας ενδιαφέρει οποιαδήποτε άλλη συνέπεια και αν έχουν.

ΟΡΙΣΜΟΣ του Execution Length
Το **μήκος** κάθε μίας **«συνολικής» I.C** που σχηματίζεται από μία ακολουθία μερικών bytes το ονομάζουμε **Execution Length**(**E.L.**). Ο σκοπός του αλγορίθμου είναι να βρίσκει τέτοια **E.L** μέσα στα Payload των πακέτων.

4.3.3 Ο Αλγόριθμος Abstract Execution(A.E.P)

Παρακάτω παρουσιάζεται ο αλγόριθμος **Abstract Execution** που στην ουσία αυτό που κάνει είναι να βρίσκει τα **E.L.** μέσα σε ένα String(byte sequence) που του δίνεται σαν παράμετρος. Ο αλγόριθμος αυτός είναι αναδρομικός και αυτό που κάνει είναι να αθροίζει μια-μια τις **E.I** που βρίσκει σε μια **I.C,** ενώ κάθε φορά που βρίσκει **jump instruction** ξανακαλεί τον εαυτό του συνεχίζοντας το ίδιο για την επόμενη **I.C.** Αυτό συνεχίζεται για κάθε πιθανό δρόμο που θα χρειαστεί να διανύσει. Ακόμα σαν παράμετρος θα του δοθεί η **αρχική θέση(pos)** από όπου θα ξεκινήσει να ελέγχει για να βρει και να μετρήσει τις **I.C**.

Εδώ πρέπει να σημειωθεί το εξής. Αν βρεθεί μέσα στο String για παράδειγμα ένα **while statement** δηλαδή ένα **jump instruction** τότε μπορεί να οδηγεί το **pos** σε προηγούμενες θέσεις του String. Ακόμα ο αλγόριθμος με βάση τις προδιαγραφές θα πρέπει να μην εκτελεί πραγματικά την εντολή άρα δεν μπορεί να ελέγξει ποιο **Branch** της εκτέλεσης να διαλέξει

παρά μόνο να ελέγξει και τα δύο Brachies όπως θα κάνει για παράδειγμα με το while statement . Αυτό έχει σαν αποτέλεσμα πολύ εύκολα ο αλγόριθμος να πέσει σε Loop. Για να αποφεύγονται τέτοιες περιπτώσεις υπάρχει ένας πίνακας που κρατά όλες τις θέσεις μέσα στο string που ελέγχθηκαν από τον αλγόριθμο ώστε αν ξαναπεράσει να εντοπιστεί ο βρόχος και να σταματήσει η εκτέλεση. Ο πίνακας ορίζουμε ότι είναι ο **visitied _pos[STRING_LEN]**.

Τέλος ως **L** ορίζουμε το σύνολο των **E.I.** που έχει βρει ο αλγόριθμος μέχρι την στιγμή που θα σταματήσει.

Αλγόριθμος:

1. Όταν στην θέση pos δεν υπάρχει **εντολή(E.I.)** επιστρέφει **L.** Αν είναι η πρώτη φορά που εκτελείται τότε το L θα έχει τιμή 0
2. Όταν η **εντολή(E.I.)** στην θέση **pos** είναι σημαδεμένη ως **Visited** στον πίνακα **visitied _pos[STRING_LEN]** τότε σημαίνει ότι εντοπίστικε **βρόχος(Loop)** και επιστρέφει **L**.
3. Βρίσκει τις **εντολές(E.I.)** ξεκινώντας από τη θέση **pos** και υπολογίζει το μήκος **L.** Παράλληλα σημαδεύει όλες τις θέσεις που έχει επισκεφτεί στον πίνακα **visitied _pos[STRING_LEN].**
4. Όταν η **I.C.** τελειώσει με **σκουπίδια(not valid E.I.)** επιστρέφει **L.**
5. Όταν η **I.C.** τελειώνει σε **Jump Instruction** τότε εκτελούνται μία από τις παρακάτω περιπτώσεις.
6. Όταν το **Jump** οδηγεί έξω από το String ή δεν μπορεί να οριστεί στατικά πού θα οδηγήσει τότε επιστέφει **L+1**.
7. Όταν το **Jump** οδηγεί μέσα στο String τότε καλείται αναδρομικά ο αλγόριθμος ξεκρινόταν με τιμή **L’.** Στο τέλος της εκτέλεσης τής ανάδρομης επιστέφει **L+L’.**
8. Όταν το **Jump** οδηγεί μέσα στο String και το Jump είναι **Conditional** τότε καλείται αναδρομικά και για τα δυο **Branchies**. Το πρώτο Branch ξεκινάει με τιμή **L’** το δεύτερο με την τιμή **L’’.** Στο τέλος της εκτέλεσης της αναδρομής για κάθε Branch **θα επιστέφει το μεγαλύτερο μεταξύ** των **L+L’** και **L+L’’.** Στην περίπτωση του δεύτερου Branch αυτό που γίνεται είναι να ξεκινάει αναδρομικά η εκτέλεση του αλγορίθμου για την άμεσο επόμενη από το **Jump** θέση με την τιμή **L’’.**

Όπως είπαμε παραπάνω το τελευταίο κριτήριο που θα μας βοηθήσει να ξεχωρίσουμε αν ένα πακέτο περιέχει B.O.E. είναι η **ποσότητα των εκτελέσιμων εντολών που εμφανίζονται μέσα σε αυτό.** Δηλαδή με άλλα λόγια θα ξεχωρίζουμε ποιο πακέτο περιέχει B.O.E με βάση τα **E.L** που θα βρίσκουμε μέσα στο πακέτο. Για το σκοπό αυτό ορίζουμε το **M.E.L.**

ΟΡΙΣΜΟΣ του Maximum Execution Length(M.E.L.):
Το **M.E.L** μίας ακολουθίας από byte(String) είναι το **μεγαλύτερο από όλα τα πιθανά E.L** που θα βρεθούν μέσα στο string **ξεκινώντας από κάθε πιθανή θέση του.** Το τελευταίο σημαίνει ότι ο αλγόριθμος θα ξεκινάει από την αρχή για κάθε **pos.**

4.3.4 Εντοπισμός του buffer overflow χρησιμοποιώντας τον αλγόριθμο A.E.P.

Με βάση όλα όσα έχουν αναφερθεί μέχρι τώρα σε αυτό το κεφάλαιο προκύπτει ότι το **M.E.L** θα είναι το κριτήριο με το οποίο θα ξεχωρίζουν τα πακέτα με B.O.E από τα υπόλοιπα φυσιολογικά πακέτα. Συγκεκριμένα με βάση την *μελέτη του sledge* παραπάνω στα πακέτα με B.O.E **<u>αναμένεται να έχουν πολύ μεγάλο M.E.L.</u>** σε σχέση με τα M.E.L των άλλων φυσιολογικών ή αλλιώς μη βλαβερών πακέτων. Αυτό οφείλεται στο ότι το Sledge είναι μια **I.C.** με πολύ μεγάλο μήκος σε σχέση με άλλες **I.C.** που κατά **«σύμπτωση»** σχηματίστηκαν.

Εύλογα θα αναρωτιόταν κανείς τι θα μπορούσε να κάνει ένας Blackhat για να ξεγελάσει τον αλγόριθμο **A.E.P** για να κρύψει την επίθεση του. Με άλλα λόγια προκύπτει το ερώτημα αν ο αλγόριθμος έχει αδυναμία και ποια είναι αυτή. Η απάντηση είναι απλή αν παρατηρήσει κανείς ότι αν προστεθούν πολλές **Jump Instraction** τότε το M.E.L που θα προκύψει θα είναι σαφώς

μικρότερο από το **M.E.L** που θα προέκυπτε αν δεν υπήρχαν τα Jump και θα ήταν ίσο με το **πραγματικό μήκος του Sledge**. Αφενός όμως με βάση την σχεδίαση του αλγόριθμου τα M.E.L που θα προκύψουν δεν θα είναι πολύ μικρά, αφετέρου το σημαντικότερο είναι ότι η τόσο έντονης μεταβολής που προκαλεί το sledge στην συμπεριφορά του payload, σε σχέση με ένα payload χωρίς sledge, είναι τόσο έντονη ώστε να **αναμένεται να εξακολουθούν να υπάρχουν πολύ μεγάλα M.E.L** ακόμα και με την χρήση του Jump.

Παρατηρώντας τις περιπτώσεις B.O.E που γίνονται μέσω δικτύου διαπιστώνεται ότι σε μία συνηθισμένη **Client-Sever** επικοινωνία τα **δεδομένα**(payload string) που ανταλλάσσονται είναι καθορισμένα από το πρωτόκολλο επικοινωνίας. Με βάση το πρωτόκολλο κάθε κομμάτι του payload έχει **συγκεκριμένο σημασιολογικό περιεχόμενο** ή πιο απλά εκεί γράφονται πληροφορίες που έχουν συγκεκριμένη χρησιμότητα και σημασία. Έτσι, η πιθανότητα να έχουν χρησιμοποιηθεί κατά **«σύμπτωση»** σε **μεγάλη ποσότητα** και **συνεχόμενα,** χαρακτήρες που αντιστοιχούν σε OP codes του επεξεργαστή μικραίνει ακόμα περισσότερο. Από αυτό προκύπτει ότι η πιθανότητα να βρεθούν μεγάλα M.E.L σε φυσιολογική κίνηση του δικτύου μικραίνει ακόμα περισσότερο. Έτσι *το M.E.L ενός πακέτου που μπορεί να περιέχει Buffer Overflow Exploit θα είναι πάντα πολύ μεγαλύτερο σε σχέση με το φυσιολογικό πακέτο τις ίδιας Client-Server επικοινωνίας.*

Η κεντρική ιδέα του εντοπισμού των επιθέσεων με τον αλγόριθμο A.E.P είναι ότι θα υπάρχει για κάθε επικοινωνία μία ανώτατη τιμή για το MEL. **Η ανώτατη αυτή τιμή θα ονομάζεται κατώφλι(Threshold)**. Κάθε **πακέτο που θα έχει M.E.L πάνω από το Threshold** θα έχει **αυξημένη πιθανότητα** να περιέχει **B.O.E**. Έτσι με τον αλγόριθμο A.E.P το πρόβλημα περιορίζεται στο πώς θα βρεθεί αυτό το Threshold και αν αυτό θα είναι κοινό για όλες τις υπηρεσίες. Το πρόβλημα θα μελετηθεί αναλυτικά παρακάτω. Προς το παρών θα θεωρούμε ότι το Threshold είναι μια σταθερή τιμή που διαχωρίζει τα B.O.E από τα άλλα πακέτα.

4.3.5 Τα προβλήματα της υλοποίησης του Αλγόριθμου Abstract Execution(A.E.P)

Ο αλγόριθμος AEP είναι αναδρομικός και αυτό έχει σαν συνέπεια να χρειάζεται αρκετή υπολογιστική ισχύ για να εκτελεστεί καθώς επίσης, αναλόγως με την είσοδο, χρειάζεται και αρκετή μνήμη. Για να μειωθεί η ανάγκη για resources σε μια υλοποίηση του αλγόριθμου θα πρέπει αυτός να ενεργοποιείται μόνο σε περίπτωση που πραγματικά είναι πιθανό να υπάρχει επίθεση από Buffer Overflow Exploit. Αυτό και μερικά άλλα προβλήματα που περιγράφονται παρακάτω είναι μερικοί από τους βασικότερους παράγοντες για μία καλή υλοποίηση του αλγορίθμου. Τα δύο βασικότερα προβλήματα εκτός από την επιλογή του Threshold είναι:

- Η βελτιστοποίησης του μηχανισμού πυροδότησης του A.E.P ώστε να μην γίνεται άσκοπη χρήση του αλγόριθμου.
- Η επιλογή του μηχανισμού που θα βρίσκει τις **E.I** διατηρώντας την μέγιστη απόδοση.

<u>4.3.5.1 Βελτιστοποίησης του μηχανισμού πυροδότησης του A.E.P ώστε να μην γίνεται άσκοπη χρήση του αλγόριθμου.</u>

Ο μηχανισμός πυροδότησης του A.E.P είναι πολύ βασικός και το πρόβλημα της επιλογής του λύνεται σχετικά εύκολα. Η κρισιμότητα του προβλήματος οφείλεται στο ότι αν δεν γίνεται έλεγχος για ποιες εισόδους ενεργοποιείται ο αλγόριθμος τότε θα κατασπαταλάται υπολογιστική ισχύ. Αυτή η σπατάλη στην καλύτερη περίπτωση θα είναι ανασταλτικός παράγοντας από τον διαχειριστή του συστήματος να ενεργοποιείσει την χρήση αυτού του αλγορίθμου θυσιάζοντας την απόδοση. Το πρόβλημα λύνεται είτε πειραματικά είτε εμπειρικά με το να μην επιτρέπουμε την ενεργοποίηση του αλγορίθμου στις περιπτώσεις που ξέρουμε ότι η πιθανότητα για Buffer Overflow Exploit είναι ελάχιστες.

Για παράδειγμα μία από τις επιλογές που θα καλύπτει ο μηχανισμός πυροδότησης είναι **να μην ενεργοποιεί** τον **A.E.P** για πακέτα που έχουν payload μικρότερου μήκους από το Threshold που έχουμε προκαθορίσει, γιατί είναι σίγουρο ότι εκεί ο αλγόριθμος δεν θα βρει B.O.E. Μια άλλη περίπτωση ειδικά για τα IDS είναι ότι δεν χρειάζεται να ελέγχονται τα πακέτα ICMP για B.O.E.

ΣΥΜΠΕΡΑΣΜΑ
Ο **μηχανισμός πυροδότησης** του A.E.P θα πρέπει να έχει **ένα σύνολο από κριτήρια** που θα τον βοηθούν να επιλέξει πότε επιτρέπει στον A.E.P να κάνει έλεγχο για B.O.E σε ένα πακέτο με σκοπό να μην κατασπαταλάται άσκοπα υπολογιστική ισχύ. Τα κριτήρια αυτά θα πρέπει να είναι αποτέλεσμα μετρήσεων ή από εμπειρία.

4.3.5.2 Η επιλογή του μηχανισμού που θα βρίσκει τις E.I διατηρώντας την μέγιστη απόδοση

Στην παραπάνω περιγραφή του αλγορίθμου αναφέρεται ότι για να αποφασιστεί, σε μία θέση **pos,** ποία θα είναι η υπόλοιπη ροή του αλγορίθμου πρέπει να αναγνωρίζεται αν υπάρχει κάποιο Op code. Η επιλογή του αλγόριθμου που θα κάνει αυτή την αναγνώριση έχει πολύ μεγάλη σημασία γιατί αφενός πρέπει να είναι γρήγορος, αφετέρου πρέπει να μην χρειάζεται πολλά resources. Μία συνηθισμένη επιλογή θα ήταν να χρησιμοποιηθεί μια κλασική διαδικασία με Hash function που θα κατακερματίσει το string στα κομμάτια των εντολών που το αποτελούν και στην συνέχεια να γίνει ανάλυση επί των κομματιών ώστε να αναγνωριστούν σε ποια Op code αντιστοιχούν. Εδώ υπάρχουν κάποια πολύ βασικά προβλήματα:

1. Η Hash function θα καθυστερεί πολύ τον αλγόριθμο A.E.P γιατί θα πρέπει πρώτα να εκτελείται αυτή και μετά ο αλγόριθμος.
2. Δεν υπάρχει συγκεκριμένος διαχωριστής μεταξύ τον Opcode άρα πρέπεί να σχηματίζει όλες τις δυνατές περιπτώσεις για κάθε **pos**.
3. Η Hash function Θα πρέπει να εκτελείται από την αρχή για κάθε νέο **pos** που θα δίνεται σαν είσοδος.

Για να αποφευχθούν τα τρία παραπάνω προβλήματα ώστε να υλοποιηθεί ο αλγόριθμος A.E.P χωρίς να ασκείται επιπλέον πίεση στο σύστημα που θα προκαλέσει μη αποδεκτή καθυστέρηση στην παρακολούθηση του δικτύου πρέπει να χρησιμοποιηθεί ένας άλλος μηχανισμός αντί της Hash Function. Οι εμπνευστές του αλγόριθμου A.E.P για να λύσουν αυτό το πρόβλημα χρησιμοποιούν τον παρακάτω μηχανισμό που θα χρησιμοποιηθεί και στην εφαρμογή αυτής της πτυχιακής.

Μηχανισμός αναγνώρισης E.I:

Για **την διαδικασία της αποκωδικοποίησης της ακολουθίας των byte(string)** ώστε να βρεθούν τα Op Codes μέσα σε αυτή χρησιμοποιείται ένα **στατικό δέντρο(Static) που ονομάζεται trie.** Το **Trie(δοκιμαστής)** είναι μια **δομή ενός ιεραρχικού δέντρου** που λειτουργεί σαν λεξικό. Το δέντρο αυτό έχει τα παρακάτω χαρακτηριστικά:

1. Κάθε κόμβος περιέχει μόνο ένα χαρακτήρα που είναι το μέρος μίας ολόκληρης λέξης που αντιστοιχεί σε κάποιο OP code.
2. Κάθε χαρακτήρας(byte) της «λέξης» αυτής αποθηκεύεται σε διαφορετικό επίπεδο του δέντρου.
3. Ο πρώτος χαρακτήρας κάθε λέξης αποθηκεύεται στον **αρχικό κόμβο του δέντρου (root node)** ενώ μαζί του αποθηκεύεται και ένας pointer(δείκτης) που δείχνει στο επόμενο επίπεδο. Για τους χαρακτήρες των επόμενων επιπέδων του δέντρου εφαρμόζεται αναδρομικά ακριβώς ο ίδιος μηχανισμός μέχρι να χτιστεί ολόκληρο το δέντρο.
4. Ο **αριθμός των χαρακτήρων** που αποτελούν ένα OP code **είναι ακριβώς ίσος με τον αριθμό των επιπέδων** που χρειάζονται για να αποθηκευτεί στο δέντρο ο OP code.
5. Τα φύλλα του δέντρου είναι η ένδειξη τέλους για κάθε «λέξη». Κάθε λέξη σχηματίζεται από μια διαδρομή(path) από την ρίζα(root) μέχρι το φύλο τερματισμού της λέξης.
6. Σε κάθε φύλο φυλάσσονται όλες οι πληροφορίες που υπάρχουν σχετικά με τον κάθε OP code. Τέτοιες πληροφορίες είναι το μήκος σε byte του OP code και οι Operands του αν έχει κτλ.

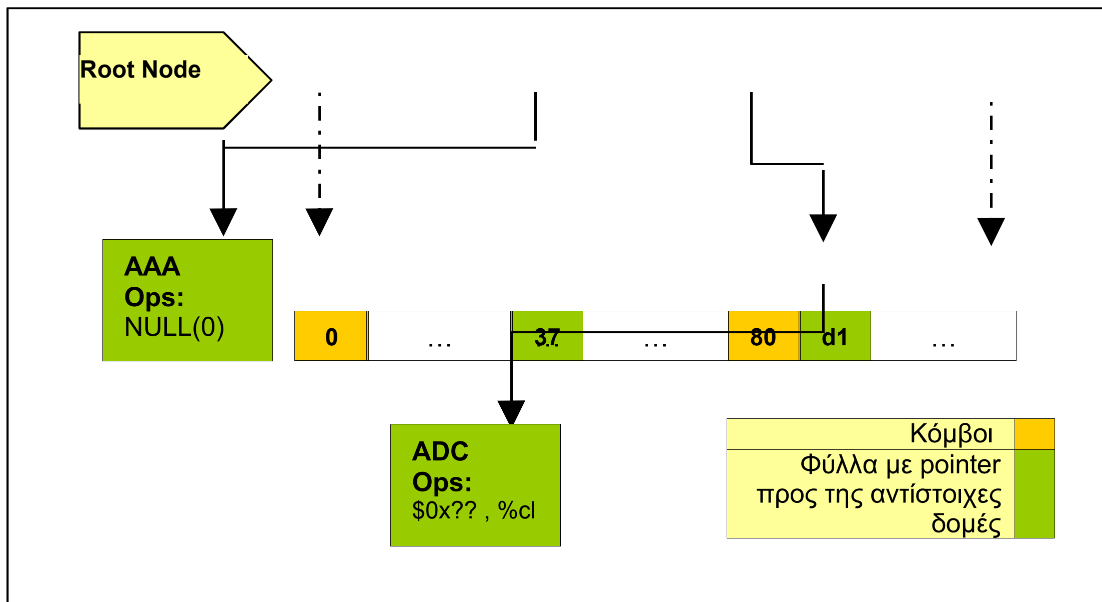
Με αυτό το δέντρο μπορεί να γίνεται πολύ γρήγορα η αναζήτηση ενός OP code σε πραγματικό χρόνο εξετάζοντας απλά ένα προς ένα τα byte. Αν η αναζήτηση καταλήξει σε φύλλο τότε αυτομάτως σημαίνει ότι βρέθηκε και ένα OP code. Έτσι **χωρίς να χρειαστεί να**

εξετάζονται όλες οι περιπτώσεις για κάθε πιθανό μήκος ενός OP code, **όπως θα γινόταν με μία hash function,** βρίσκουμε αμέσως την λέξη δηλαδή την **E.I**. Ακόμα επειδή στα φύλα υπάρχουν πληροφορίες για τα Operands του κάθε OP code μπορεί πολύ εύκολα και άμεσα χωρίς περαιτέρω επεξεργασία να βρεθούν πόσα από τα επόμενα bytes είναι Operands του. Έτσι μπορεί **ο δείκτης** που δείχνει την αρχή του OP code **να μετακεινηθεί** τόσα Byte όσα τα byte που αποτελούν τους Operands του έτσι ώστε να τα **παρακάμψει(Skip)** ή για να γίνει η **περαιτέρω επεξεργασία** τους.

ΣΗΜΑΝΤΙΚΗ ΠΑΡΑΤΗΡΗΣΗ
Για να μπορεί να γίνεται όσο το δυνατόν πιο αποτελεσματική αναζήτηση του Sledge ενός B.O.E. θα πρέπει μέσα στο **Trie δέντρο να υπάρχει ολόκληρο το Instruction Set** του εκάστοτε επεξεργαστή. Φυσικά αυτό σημαίνει ότι ο ίδιος αλγόριθμος A.E.P θα πρέπει να τρέξει για κάθε ακολουθία από byte(string) τόσες φορές **όσα είναι τα δέντρα που αντιστοιχούν σε κάθε επεξεργαστή**.

Στην παρακάτω εικόνα φαίνετε μια μικρογραφία του **Trie δέντρου** για το επεξεργαστή Pentium I της Intel.

Δείγμα της δομής του Trie δέντρου



Σχήμα 4.3

4.4 Η αναζήτηση των πιθανών Return Addressees ως μηχανισμός επιβεβαίωσης (Validation) για τον αλγόριθμο εντοπισμού των B.O.E

4.4.1 Γενικά

Ο αλγόριθμος A.E.P όπως και κάθε μέθοδος **Anomaly-Based Detection** ανάγει το πρόβλημα του εντοπισμού των B.O.E σε ένα **συνηθισμένο πρόβλημα σωστής επιλογής κατωφλιού (Threshold)** δηλαδή αυτής ή αυτών των τιμών που θα διαχωρίσουν τα **«φυσιολογικά»** πακέτα από τα **μη φυσιολογικά.** Η επιλογή αυτής της τιμής είναι ιδιαίτερα δύσκολη και συνήθως η λύση του πρόβληματος περιορίζεται στο να βρεθεί μια αρκετά καλή τιμή που δεν θα προκαλεί ούτε πολλά False Positive (βλέπε σελ.114) ούτε πολλά False Negative (βλέπε σελ.114). Το πώς θα γίνει η επιλογή του Threshold θα αναλυθεί παρακάτω.

Όσο καλά και να είναι τα αποτελέσματα του A.E.P με κάποιο Threshold αυτός δεν παύει να έχει την αδυναμία που έχουν όλοι οι μηχανισμοί **Anomaly-Based Detection** δηλαδή **δεν**

μπορεί ποτέ να εντοπίζει όλα τα B.O.E. Για να μειωθεί το πρόβλημα υπάρχουν δύο λύσεις. Η μία είναι να εφαρμόσουμε τον αλγόριθμο σαν module για κάθε εφαρμογή όπως αρχικά εφαρμόσθηκε από τον T.Toth. Η δεύτερη λύση είναι **να βρούμε ένα μηχανισμό που θα επιβεβαιώνει** ότι ο A.E.P εντόπισε ένα πραγματικό B.O.E. Η ανάγκη για έναν τέτοιο μηχανισμό δεν προκύπτει μόνο από την θέληση μας να έχουμε μια ακόμα ποιο αξιόπιστη μηχανή που θα εντοπίζει τις επιθέσεις, αλλά γιατί αυτή την αδυναμία μπορεί ο ίδιος ο Blackhat να την χρησιμοποιήσει για να κρύψει την επίθεση του. Αν για παράδειγμα υποθέσουμε ότι έχουμε ένα NIDS που εκτελεί αυτόν τον αλγόριθμο για να εντοπίζει τα BOE τότε πολύ εύκολα ο Blackhat μπορεί να στέλνει πολλά πακέτα που να περιέχουν μεγάλα Sledge χωρίς όμως να στέλνει και το πραγματικό Exploit προκαλώντας ψευδή συναγερμό. Ο ψευδής αυτός συναγερμός θα προκαλέσει σύγχυση στον Administrator που θα βλέπει ότι το NIDS τον ειδοποιεί για BOE που δεν συμβαίνουν και εκμεταλλευόμενος αυτή την σύγχυση κάπου εκεί ανάμεσα στα χιλιάδες ψεύτικα πακέτα θα κρύβεται και το πραγματικό Exploit. Τέτοια μορφή επίθεσης την ονομάζουμε **Decoy Attack** και για να την αποφεύγουμε **χρειάζεται ένας Μηχανισμός Validation**.

Στην **παράγραφο 4.2.3** διαπιστώσαμε ότι η **αναζήτηση των RA** μέσα σε ένα πακέτο δεν είναι καλός μηχανισμός για εντοπισμό, αλλά είναι όμως ένας καλός βοηθητικός **μηχανισμός επιβεβαίωσης (Validation) εντοπισμού B.O.E.** Για αυτό το λόγο μία καλή πρόταση για **μηχανισμό Validation είναι να γίνεται αναζήτηση των RA**. Αυτή η πρόταση θα εφαρμοστεί και στο πρόγραμμα που θα υλοποιηθεί στα πλαίσια αυτής της πτυχιακής εργασίας. Η εφαρμογή αυτού του μηχανισμού έχει και αυτή τις δυσκολίες της αλλά σίγουρα όχι τόσες πολλές όσο ο A.E.P.

4.4.2 Προδιαγραφές αλγορίθμου εντοπισμού(Finding of) των R.A

Όπως είπαμε και στην προηγούμενη παράγραφο τα προβλήματα που κυρίως υπάρχουν για να βρεθεί το RA είναι αφενός ότι πρέπει να ληφθούν υπόψη όλες οι περιπτώσεις περιοχών μνήμης που πιθανόν να χρησιμοποιεί κάθε εφαρμογή, αφετέρου πρέπει να λαμβάνονται υπόψη όλες οι πιθανές παραλλαγές του RA. Από αυτά προκύπτουν οι έξεις προδιαγραφές:

- Η αναζήτηση πρέπει να γίνεται κάθε 4 byte ώστε οι διευθύνσεις να συμπίπτουν με i386 μηχανισμό διευθυνσιοδοτησης. Για άλλο μηχανισμό θα χρειαστεί και άλλη επιλογή σε αριθμό Byte.
- Στο άθροισμα των Return Address που θα βρίσκονται από τον αλγόριθμο **θα πρέπει να προστίθενται μόνο** οι διευθύνσεις μνήμης που ανήκουν στην περιοχή της στοίβας ή σε άλλες περιοχές που ένα B.O.E μπορεί να οδηγήσει επιτυχώς τον IP (Instating Pointer) χωρίς να ενεργοποιήσει κάποιο μηχανισμό προστασίας σφαλμάτων του λειτουργικού συστήματος. Τέτοιοι μηχανισμοί προστασίας του λειτουργικού συστήματος είναι αυτοί που ενεργοποιούνται όταν εμφανίζεται το μηνύμα Segmentation Fault ή illegal operation κτλ.
- Στο παραπάνω άθροισμα θα πρέπει να λαμβάνονται υπόψη και οι παραλλαγές ενός RA που μπορεί να υπάρχουν στο B.O.E, από τον Blackhat, για λόγους αντιπερισπασμού. Οι παραλλαγές μπορεί να είναι είτε **RA-x** είναι **RA+x**. Το **x** είναι ένα μικρό Offset από το αρχικό RA για παράδειγμα οι τιμές **2, 4, 8, 12, κτλ**.

4.4.3 Ο Αλγόριθμος Validation via Findig RA (V.F.R.A)

Ο αλγόριθμος αυτός είναι αναδρομικός και αρχίζει να εκτελείται για κάθε πιθανή θέση του String. Αυτό που κάνει είναι να αρχίζει πάντα από μία θέση **pos** και να συνεχίζει ανά 4 byte να βρίσκει RA που βρίσκονται σε μια συγκεκριμένη περιοχή μνήμης. Αν βρει μετά από το πρώτο RA ένα ακόμη RA με ακριβώς την ίδια τιμή ή μια μικρή παραλλαγή του, για παράδειγμα RA+16, τότε αυξάνει έναν αθροιστή που μετρά πόσα RA έχει βρεί ο αλγόριθμος.

1. Παίρνει από την θέση **pos** 4 byte.
2. Τα 4 byte τα αντιγράφει σε μια αριθμητική μεταβλητή 4byte (στην C long).
3. Ελέγχει **αν ο αριθμός που προκύπτει ανήκει** στην στοίβα του συστήματος ή αν ανήκει στην περιοχή μνήμης μου φορτώνονται τα DLL ειδικά στην περίπτωση των Ms Windows.
4. Αν **ανήκει** και ο αλγόριθμος εκτελέστικε πρώτη φορά τότε φυλάσσεται η τιμή αυτή στο **Saved_RA** και αυξάνεται ο αθροιστής **RA_addresses** που κρατά τον αριθμό των RA που βρέθηκαν κατά 1.

5. Αν **ανήκει** και ο αριθμός είναι ίδιος με το **Saved_RA** ή μεγαλύτερο κατά **x** ή μικρότερος κατά **x** τότε ο **RA_addresses** αυξάνεται κατά ένα. Το **x** είναι το μέγιστο Offset που μπορεί να υπάρξει και το B.O.E να εξακολουθεί να είναι επιτυχημένο.
6. Στο τέλος του αλγορίθμου επιστέφεται το **RA_addresses.**

> **ΣΗΜΕΙΩΣΗ**
> Ο αλγόριθμος είναι αναδρομικός και το **RA_addresses** που θα επιστραφεί θα είναι το άθροισμα όλων των αναδρομικών κλίσεων του αλγορίθμου.

4.4.4 Τα προβλήματα της υλοποίησης του Αλγόριθμου V.F.R.A και η λύση τους

Όπως και στην περίπτωση του AEP έτσι και εδώ όταν θα υλοποιηθεί ο αλγόριθμος αυτός θα αντιμετωπιστούν τα παρακάτω προβλήματα:

- Η επιλογή του **Threshold.**
- Η «σύμπτωση» που προκαλεί False Positive.

4.4.4.1 Η επιλογή του Threshold

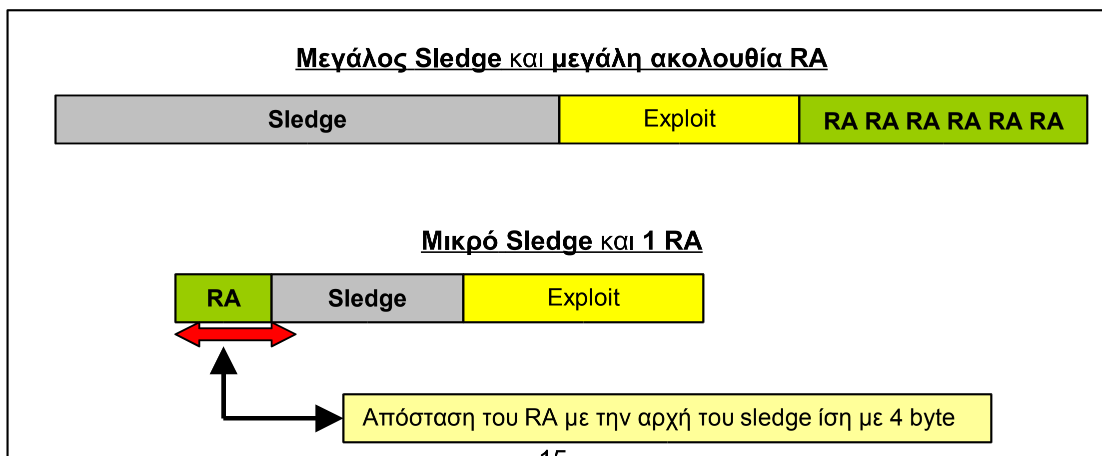
Όπως και στην περίπτωση του A.E.P για το sledge έτσι και στην περίπτωση του V.F.R.A για την RA πρέπει να υπάρχει ένα κατώφλι(Threshold) που θα ξεχωρίζει τα B.O.E από τα φυσιολογικά πακέτα. Ειδικά όμως για την περίπτωση του RA όπως είπαμε σε αντίθεση με το Sledge αρκεί να υπάρχει έστω και **μια μόνο** φορά το RA στην κατάλληλη θέση για να πετύχει το Exploit. Στην περίπτωση λοιπόν αυτή το Threshold **θα πρέπει να είναι αρκετά** μικρό. Η επιλογή του Threshold θα γίνεται σε συνάρτηση με το Threshold που θα χρησιμοποιείται από τον A.E.P.

4.4.4.2 Η «σύμπτωση» που προκαλεί False Positive

Όπως αναφέρθηκε πολλές φορές παραπάνω η πιθανότητα κατά «σύμπτωση» μια ακολουθία από bytes να αντιστοιχεί σε Op code, ενώ αυτή δεν είναι μέρος του Sledge, είναι αρκετά μεγάλη. Το ίδιο ισχύει και για την **πιθανότητα ένα Op Code που είναι μέρος του Sledge να αντιστοιχεί κατά «σύμπτωση» σε ένα RA.** Έτσι, ενώ μπορεί να ικανοποιούνται όλα τα κριτήρια που συγκλίνουν στο συμπέρασμα ότι βρέθηκε μια ακολουθία από RA τελικά δεν θα είναι τίποτα άλλο από το Sledge που προηγουμένως ο A.E.P εντόπισε. Για να λυθεί αυτό το πρόβλημα επιστρατεύεται **η έννοια της «Γεωμετρίας» του B.O.E**

4.4.5 Η «Γεωμετρία» του πακέτου που περιέχει Buffer Overflow Exploit ως κριτήριο για την αποφυγή False Positive.

Όπως προαναφέρθηκε στην μελέτη των BOE όλα έχουν το κοινό χαρακτηριστικό ότι αποτελούνται από τρία μέρη: **το Sledge, το Shellcode(ή Exploit Code)** και **το κομμάτι με τις RA.** Ενώ όπως είπαμε η σειρά που θα υπάρχουν αυτά δεν είναι απόλυτη, τουλάχιστον μας εξασφαλίζει ότι **η αρχή του Sledge** με **το πρώτο byte της ακολουθίας των RA** θα έχουν τουλάχιστον ένα RA(4 byte) απόσταση. Αυτό φαίνεται καθαρά αν εξετάσουμε τις δύο πιο ακραίες περιπτώσεις:



Σχήμα 4.4

Στην πρώτη περίπτωση διαπιστώνουμε ότι το πρώτο RA έχει τουλάχιστον τόση απόσταση σε Byte όσο είναι η τιμή του Threshold(Threshold για την ανεύρεση του sledge). Στην δεύτερη περίπτωση έχει τουλάχιστον τόση απόσταση όσο είναι η τιμή του **RA_Threshold**(Threshold που ξεχωρίζει τα πακέτα που έχουν **ακολουθίες RA)**. Η χειρότερη περίπτωση όπως φαίνεται και στο παράδειγμα είναι το μήκος της ακολουθίας να είναι **1** (μια) RA αφού χρησιμοποιούνται μόνο 4 Byte. Αυτή όμως την περίπτωση δεν θα την βρει ο **V.F.R.A** γιατί το RA_Threshold είναι σχεδόν απίθανο να έχει τιμή 1 όπως θα δούμε και παρακάτω.

4.4.6 Συμπεράσματα για τον μηχανισμό Validation.

Ο μηχανισμός επιβεβαίωσης(validation) ότι ο A.E.P εντόπισε ένα πραγματικό B.O.E είναι ιδιαίτερα σημαντικός. Για να είναι όμως αποτελεσματικός και χρήσιμος θα πρέπει να συνδυάζει τον αλγόριθμο V.F.R.A με την αναζήτηση της «γεωμετρίας» του πακέτου ώστε να εξασφαλίσει ότι **δεν θα κάνει λάθος εκτίμηση** λόγω της αδυναμίας που έχει ο V.F.R.A. Τελικά αυτό που θα παίρνουμε σαν έξοδο από τον μηχανισμό επιβεβαίωσης είναι μία τιμή, ένα ποσοστό επί τις εκατό, που θα δείχνει την πιθανότητα το πακέτο που εντόπισε ο A.E.P να περιέχει πράγματι ένα Buffer Overflow Exploit.

Ένα τέτοιο ποσοστό που παράγεται από τον V.F.R.A εξαρτάται από τρεις παράγοντες:

1. Από το ποσοστό επιτυχίας που έχει ο A.E.P για την εκάστοτε Client-Server επικοινωνία.
2. Από την ύπαρξη ενός συνόλου από RA πάνω από το RA_Threshold.
3. Από την απόσταση είτε αρνητική είτε θετική της **αρχής του Sledge** από την **αρχή της ακολουθία RA.**

Μία απλή υλοποίηση ενός μηχανισμού Validation που χρησιμοποιεί τα αποτελέσματα του V.F.R.A παρουσιάζεται παρακάτω :

1. Αποθύκευση της θέσης **pos** που ο A.E.P βρήκε την **αρχή του Sledge** στην μεταβλητή **BO_pos**.
2. Σώσιμο της θέσης **pos** που ο V.F.R.A βρήκε το **πρώτο RA** στην μεταβλητή **RA_pos**
3. Αύξηση του ποσοστού **Validation** κατά **x** αναλόγως με την Client-Server επικοινωνία. Το **x** θα πρέπει να είναι αποτέλεσμα της μέτρησης της αξιοπιστίας του AEP για μία τέτοια επικοινωνία για παράδειγμα σε μια HTTP επικοινωνίας(HTTP request).
4. Αύξηση του ποσοστού **Validation** κατά **Y** λόγο της ύπαρξης σημαντικού αριθμού RA.
5. Αύξηση του ποσοστού **Validation** κατά **x** αν η απόσταση είναι θετική ή **z** αν η απόσταση είναι αρνητική. Αν η **απόσταση είναι 0** τότε αφαιρείται το **Y** του βήματος 4 γιατί το sledge κατά «σύμπτωση» το αναγνώρισε και σαν ακολουθία από RA.

4.5 Η επιλογή του Threshold, το τελικό βήμα πριν την υλοποίηση

4.5.1 Γενικά

Ένα **Buffer Overflow Exploit,** όπως είπαμε εξαρτάται από πολλούς παράγοντες όπως από το λειτουργικό σύστημα, από τον επεξεργαστή και πολλούς άλλους. Με τον μηχανισμό **Abstract Execution of payload** καταφέραμε να ανάγουμε το πρόβλημα σε αυτό της επιλογής του κατάλληλου **Threshold** το οποίο είναι το κριτήριο διαχωρισμού των πακέτων που περιέχουν B.O.E. Η επιλογή αυτή του Threshold ενώ μειώνει κατά πολύ την δυσκολία του προβλήματος εντοπισμού(Detection) εξακολουθεί να είναι αρκετά πολύπλοκο γιατί και το ίδιο το **Threshold** εξαρτάται από πολλούς παράγοντες:

- Εξαρτάται από το εκάστοτε πρωτόκολλο της Client-Server επικοινωνίας.
- Εξαρτάται από την ίδια την υλοποίηση του διαχειριστή του πρωτοκόλλου συνήθως σε επίπεδο Application(OSI Layer).

- Εξαρτάται από το Buffer της ίδιας της εφαρμογής που θα εκμεταλλευτεί το B.O.E.
- Εξαρτάται από τις τυχόν βιβλιοθήκες που θα χρησιμοποιεί η εφαρμογή και που αυτές με την σειρά τους θα κληροδοτούν τις Buffer Overflow αδυναμίες τους σε αυτήν.

Από τα παραπάνω οι δύο πρώτες περιπτώσεις καθορίζουν το **μέγιστο μήκος των E.L.** που μπορούν να βρεθούν μέσα σε ένα πακέτο ώστε να θεωρηθούν ως **μη επικίνδυνα**. Τα δύο τελευταία είναι αυτά που θα καθορίσουν **την ελάχιστη τιμή** που χρειάζεται να είναι το μήκος Sledge μίας εφαρμογής ώστε το ΒΟΕ να πετύχει όταν αυτό σταλθεί εναντίον της.

4.5.2 Εξαρτάται από το εκάστοτε πρωτόκολλο της Client-Server επικοινωνίας

4.5.2.1 Γενικά

Όπως είναι γνωστό σε κάθε Client-Server εφαρμογή καθορίζεται ένα πρωτόκολλο που εξασφαλίζει την επικοινωνία μεταξύ των δύο πλευρών. Το πρωτόκολλο καθορίζει το μέγεθος της πληροφορίας που θα μεταφέρεται με κάθε πακέτο και την σημασία της. Επίσης στα πρωτόκολλα συνήθως υπάρχουν δύο είδη πληροφορίας που ανταλλάσσονται. Το πρώτο είναι τα δεδομένα που τελικά αυτά χρειάζεται να μεταφερθούν και το δεύτερο είναι οι εντολές που συνήθως ο Client με αυτές περιγράφει στον Server πώς αυτός πρέπει να χειριστεί τα δεδομένα ή τι ειδική σημασία έχουν αυτά. Για παράδειγμα στην Telnet επικοινωνία κάθε φορά που εμφανίζεται ένας ειδικό χαρακτήρας τότε όλοι οι υπόλοιποι χαρακτήρες θεωρούνται εντολές. Τα δεδομένα μπορεί να διακινούνται μαζί με τις εντολές πάνω από το ίδιο κανάλι. Ενώ στην περίπτωση του FTP πρωτοκόλλου άλλο είναι το κανάλι επικοινωνίας μεταξύ Client και Server για τις εντολές και άλλο για τα δεδομένα. Τέλος στην περίπτωση του HTTP το URL είναι αυτό που καθορίζει το αίτημα(request) του Client προς τον server ενώ πριν από αυτό μέσα στο πακέτο υπάρχει μία σχετική εντολή όπως GET ή POST που καθορίζουν το πώς ο Server θα χειριστεί το URI που τελικά θα λάβει.

Έτσι για κάθε πρωτόκολλο χρειάζεται να γνωρίζουμε πόσα κανάλια χρησιμοποιεί για την επικοινωνία μεταξύ των δύο πλευρών ή ποιά άλλη ιδιαιτερότητα έχει. Όταν **λέμε κανάλια εννοούμε ποιές TCP ή UDP ports** χρησιμοποιεί ο Server και τι είδους πληροφορίες δέχεται εκεί. Αυτό είναι ιδιαίτερα σημαντικό να το γνωρίζουμε **όχι μόνο για να καθορίσουμε το Threshold αλλά κυρίως για να ξέρουμε σε ποιά περίπτωση αναμένουμε B.O.E και σε ποία η πληροφορία μας είναι αδιάφορη**.

4.5.3.2 Συμπέρασμα

Η γνώση το πρωτοκόλλου λοιπόν καθορίζει το ποίο είναι το μεγαλύτερο μήκος πληροφορίας που μας ενδιαφέρει και θα χειριστεί ο Server αλλά και σε ποια TCP/UDP πόρτα το αναμένουμε. Το Threshold δεν θα πρέπει να είναι μεγαλύτερο από το μέγιστο μήκος γιατί μην ξεχνάμε ότι για να γίνει η υπερχείλιση βοηθάει και το μήκος του RA αλλά και αυτό του Shellcode. Επίσης μας δίνει μια πολύ καλή ένδειξη αν περιμένουμε ένα σχετικά μεγάλο ή μικρό sledge. Φυσικά όσο μεγαλύτερο είναι το Buffer που θα δεχτεί τα δεδομένα τόσο πιθανότερο είναι να βρεθεί μεγάλο Sledge.

4.5.3 Εξαρτάται από την ίδια την υλοποίηση του διαχειριστή του πρωτοκόλλου συνήθως σε επίπεδο Application

Το σημαντικότερο κριτήριο της επιλογής του Threshold είναι ο τρόπος υλοποιησης του κομματιού μιας εφαρμογής που διαχειρίζεται το πρωτόκολλο επικοινωνίας. Αυτό συμβαίνει γιατί αν το κομμάτι της εφαρμογής που λειτουργεί σαν διαχειριστής του πρωτοκόλλου δεν έχει στατικό Buffer τότε δεν χρειάζεται να προστατεύουμε αυτή την εφαρμογή αφού έχει δικό της μηχανισμό που εξασφαλίζει ότι δεν θα συμβεί B.O.E. Το ίδιο ισχύει αν αυτο το κομμάτι της εφαρμογής ελέγχει το μήκος της πληροφορίας που θα λάβει ή αν βάζει στο Buffer συγκεκριμενο αριθμό από byte άνεξάρτητα από το συνολικό μήκος της πληροφορίας που έφτασε. Στις περισσότερες περιπτώσεις όμως μία τέτοια λειτουργία την αναλαμβάνει ένα κομμάτι κώδικα που έχει γραφτεί εδώ και πολλά χρόνια και ενδεχομένως να είναι μέρος μιας συνηθισμένης βιβλιοθήκης γραμμένης σε C. Στην περίπτωση που γνωρίζουμε ακριβώς τη αδυναμία ή που γνωρίζουμε το είδος των B.O. Exploit που χρησιμοποιούνται για αυτές τότε

μπορούμε να βρούμε μία **πολλή καλή τιμή για το Threshold και για την συγκεκριμένη υπηρεσία** π.χ. τα exploits που χρησιμοποιούνται προς τον IIS.

ΣΥΜΠΕΡΑΣΜΑ
Η γνώση του μηχανισμού διαχείριση που υλοποιείται σε κάθε εφαρμογή με σκοπό την διαχείριση του Application Layer πρωτοκόλλου της εκάστοτε εφαρμογής μπορεί να μας δώσει μία καλή πληροφορία για την τιμή του Threshold που θα χρησιμοποιήσουμε. Το Threshold αυτό συνήθως θα είναι ίσο με το μήκος του Sledge των BOE που κατά μέσο όρο χρησιμοποιείται εναντίον της εφαρμογής αυτής. Αν η εφαρμογή είναι Open Source τότε θα χρειαστεί να ελεχθεί ο κώδικας αυτού του μηχανισμού και να βγουν τα κατάλληλα συμπεράσματα ανάλογα με το μήκος του αδύναμου Buffer του μηχανισμού και των συναρτήσεων που το χειρίζονται. ____

4.5.4 Εξαρτάται από το Buffer της ίδιας της εφαρμογής που θα εκμεταλλευτεί το B.O.E.

Αν η αδυναμία προκύπτει από την ίδια την εφαρμογή και όχι από τον μηχανισμό Decoding που περιγράψαμε παραπάνω τότε αυτό που πρέπει να γνωρίζουμε είναι το μέγεθος του αδύναμου buffer ώστε να εκτιμήσουμε το μήκος του sledge που κατά μέσο όρο θα χρησιμοποιηθεί. Στην περίπτωση αυτή όπως και στις προηγούμενες το Threshold δεν πρέπει να είναι μεγαλύτερο από το Buffer **αλλά αναλογικά ούτε πολύ μικρότερο**.

4.5.6 Εξαρτάται από τις τυχόν βιβλιοθήκες που θα χρησιμοποιεί η εφαρμογή που αυτές με την σειρά τους θα κληροδοτούν τις Buffer Overflow αδυναμίες τους σε αυτήν.

Όπως είπαμε και στα προηγούμενα κεφάλαια είναι πολλές οι αδυναμίες που εμφανίζουν εφαρμογές που είναι σχετικά καινούργιες και ενδεχομένως έχουν προβλέψει περιπτώσεις επιθέσεων που οφείλονται σε BOE. Οι **αδυναμίες αυτές οφείλονται σε βιβλιοθήκες** που συνήθως χρησιμοποιούνται από όλες σχεδόν τις εφαρμογές. Δεν είναι λίγες οι περιπτώσεις εφαρμογών στα Ms Windows που παρουσιάζουν τις ίδιες αδυναμίες μεταξύ τους και που αυτές οφείλονται σε **DLL**. Ακόμα δεν είναι λίγες οι αδυναμίες που εμφανίζονται λόγω της χρήσης βιβλιοθηκών που περιέχουν συνηθισμένες βιβλιοθήκες της C όπως η scanf και πολλές άλλες όπως περιγράφεται αναλυτικά στη **παράγραφο 3.3.2** του **κεφαλαίου 3** .

ΣΥΜΠΕΡΑΣΜΑ
Στην περίπτωση των βιβλιοθηκών όπως και στις παραπάνω περιπτώσεις το Threshold θα καθορίζεται από την Buffer Overflow αδυναμία της κάθε μίας βιβλιοθήκης που χρησιμοποιεί η εφαρμογή. Η τιμή του Threshold θα πρέπει να είναι μικρότερη από το μικρότερο αδύναμο(Vulnerable) Buffer που έχει κληρονομήσει η εφαρμογή από όλες τις βιβλιοθήκης που χρησιμοποιεί.

4.5.7 Συμπέρασμα για την επιλογή του Threshold

Το συμπέρασμα από όλα τα παραπάνω είναι ότι ακόμα και αν γνωρίζουμε με ακρίβεια το μέγεθος του Buffer που μπορεί να εκμεταλλευτεί ο Blackhat δεν μπορούμε να ξέρουμε ακριβώς το ελάχιστο δυνατό Sledge που μπορεί να χρησιμοποιήσει. Από την άλλη ακόμα και αν ξέρουμε με βάση το πρωτόκολλο πώς ακριβώς δομείται η επικοινωνία μεταξύ Client και Server, δεν μπορούμε να ξέρουμε ακριβώς ποιο είναι το μέγιστο **E.L** που μπορούμε να συναντήσουμε χωρίς αυτό να είναι μέρος ενός B.O.E.

Για να μπορέσουμε να βρούμε το Threshold πρέπει και για τούς τέσσερεις παραπάνω παράγοντες να γίνουν πειραματικές δοκιμές και να εξεταστούν τα **E.L** που προκύπτουν.

ΣΗΜΑΝΤΙΚΗ ΠΑΡΑΤΗΡΗΣΗ
Το Threshold με βάση τα παραπάνω θα πρέπει να είναι σημαντικά μεγαλύτερο από το μεγαλύτερο E.L που δεν οφείλεται σε Buffer Overflow Exploits και αρκετά μικρό ώστε να είναι μικρότερο από το μικρότερο αδύναμο Buffer της συγκεκριμένης εφαρμογής.

4.6 Πειραματικά δεδομένα για την επιλογή του Threshold και η διαφορά υλοποίησης από Apache module σε IDS.

4.6.1 Γενικά

Οι εμπνευστές του αλγορίθμου A.E.P λαμβάνοντας υπόψη τους όλες τις παραπάνω δυσκολίες προτείνουν ότι η υλοποίηση του Αλγορίθμου θα πρέπει να γίνεται σαν μηχανισμός ελέγχου για κάθε εφαρμογή που θέλουμε να προστατέψουμε. Αυτή η επιλογή έχει σαν αποτέλεσμα να χρειάζεται πειραματικός έλεγχος μόνο για την συγκεκριμένη εφαρμογή που θα χρειαστεί να προστατέψει ώστε να επιλεγεί το Threshold με μεγάλη επιτυχία. Σαν δεύτερη συνέπεια έχει ότι μπορεί να χρειαστεί να χρησιμοποιείται ο μηχανισμός Decoding που έχει η ίδια η εφαρμογή. Με βάση αυτή την επιλογή οι T.Toth και C.Kruegel έκαναν το παρακάτω πείραμα.

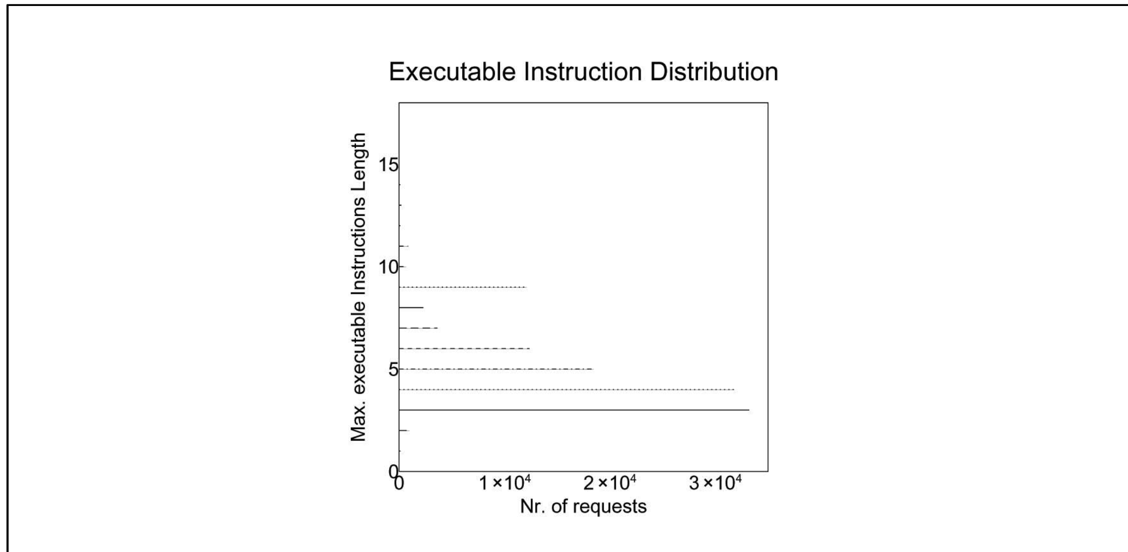
4.6.2 Πειραματικά δεδομένα για την επιλογή του Threshold

Ο αλγόριθμος υλοποιήθηκε σαν module του Apache Server 1.3 πού είναι ο γνωστότερος HTTP Server πού χρησιμοποιείται σε Unix Like συστήματα. Για να μπορεί να γίνεται σωστός έλεγχος για τυχόν B.O.E το module αυτό ενεργοποιείται για κάθε request αμέσως μετά τον μηχανισμό decoding. Φυσικά ο μηχανισμός decoding έχει εξασφαλιστεί ότι δεν έχει Buffer Overflow αδυναμίες. Το πείραμα έγινε σε εργαστηριακές συνθήκες στο web server του εργαστηρίου των T.Toth και C.Kruegel.

Αυτά που μετρήθηκαν ήταν:

1. Τα **μέγιστο M.E.L.** από όλα τα requests που έγιναν προς τον Server και δεν ήταν Buffer Overflow. Ακόμα για να εξασφαλιστεί ότι δεν θα γίνει λάθος στην επιλογή του Threshold δεν λήφθηκαν υπόψη τα request που δεν ήταν «κατανοητά» ή «νόμιμα» από τον Server και θα τα απέρριπτε πριν καν τα εξετάσει.
2. Ο **μέσος όρος των M.E.L. των sledge** από B.O.E προς τον Apache Server αλλά και των επιθέσεων πρός άλλους Server όπως ο IIS.
3. Για λόγους επιβεβαίωσης ότι ο ίδιος αλγόριθμος θα μπορούσε να δουλέψει και για άλλες υπηρεσίες το ίδιο καλά και ενδεχομένως με το ίδιο threshold χρησιμοποιήθηκε και ένα σύνολο από επιθέσείς που απευθύνονται σε FTP υπηρεσία.

Στο παρακάτω διάγραμμα φαίνεται η κατανομή τον M.E.L σε ένα σύνολο από requests που δεν έχουν B.O.E. :



Από το παραπάνω διάγραμμα προκύπτει ότι από 30.000 requests:

- Μόνο 350 είχαν M.E.L 0.
- Τα περισσότερα πακέτα είχαν M.E.L 3 με 4.
- Το **μέγιστο MEL** που βρέθηκε είχε την τιμή 16.

Στον παρακάτω πίνακα φαίνεται ο μέσος όρος των μεγεθών του sledge που βρέθηκαν σε πακέτα που είχαν B.O.E.

Όνομα του Exploit	*M.E.L*
IIS 4 hack 307	***591***
IIS 2000	***635***
Wu-ftpd/2.6-id1387	***251***

Με βάση τον παραπάνω πίνακα το M.E.L των requests που είχαν B.O.E είναι **σημαντικά μεγαλύτερο από τα άλλα**. Αυτό με **βάση τα παραπάνω ήταν αναμενόμενο** και επιβεβαιώνει την υπόθεση πάνω στην οποία στηρίχτηκε ο Αλγόριθμος. Όπως φαίνεται παραπάνω το M.E.L κυμαίνεται από 250 έως 600 περίπου. Αυτό που πρέπει να λάβουμε υπόψη μας είναι ότι η διαφορά από 16 που είναι το μεγιστο M.E.L ως 250 που είναι το ελαχιστο Sledge είναι **πολύ μεγάλη** κάτι που μας εξασφαλίζει ότι οποιαδήποτε τιμή μεταξύ των δύο ορίων είναι αρκετά καλή. Όμως πρέπει να λάβουμε υπόψη μας ότι ο Blackhat μπορεί για να κρύψει την επίθεση του από μία τέτοια εφαρμογή να χρησιμοποιεί sledge μικρότερο από αυτό του Threshold. Από την άλλη μια πολύ μικρή τιμή θα προκαλέσει πιθανόν πολλά Faulse Positive. Αρα πρέπει να έιμαστε πολύ προσεχτικοί στην επιλογή του Threshold.

4.7 Η επιλογή του Threshold

4.7.1 Η μαγική τιμή για το Threshold

Η **τιμή του threshold** στην περίπτωση του Apache module επιλέχθηκε να είναι το **30**. Η τιμή αυτή είναι **η μαγική τιμή** που ψάχναμε να βρούμε ώστε να μπορεί ο μηχανισμός εντοπισμού να ξεχωρίζει τα Buffer Overflow Exploits από τα άλλα πακέτα. Η επιλογή αυτή έγινε για τους λόγους που περιγράψαμε παραπάνω. Έτσι η τιμή του Threshold επιλέχθηκε αφενός αρκετά μακριά από το μεγαλύτερο M.E.L που βρέθηκε σε φυσιολογικό Traffic(στην προκείμενη περίπτωση το **διπλάσιο**) αφετέρου αρκετά μικρότερη από το μικρότερο M.E.L ενός B.O.E. Η τιμή αυτή θα αναγκάσει το Blackhat να βρει ένα τρόπο για να μπορέσει να μειώσει το sledged κάτω από την τιμή του 30. Αυτό όμως είναι πολύ δυσκολο να το καταφέρει ακόμα και αν προσθέσει πολλά **Jump Instructions μεσα στο Sledge.** Για παράδειγμα δεν θα καταφέρει να

μειώσει την τιμή από 250 σε 30 στην περίπτωση του **BOE "Wu-ftpd/2.6-id1387"**. Για την περίπτωση του HTTP Service με αρκετά μεγάλη επιτυχία καταφέραμε να βρούμε μια τιμή για το Threshold ενώ θα δούμε παρακάτω πώς έγινε αυτή η επιλογή.

Την ίδια διαδικασία θα πρέπει να κάνουμε για κάθε μια δικτυακή υπηρεσία ώστε να βρούμε το κατάλληλο threshold. Αυτό όμως είναι αρκετά επίπονη διαδικασία και απαιτεί πολύ χρόνο. Αυτή είναι η βασική δυσκολία που πρέπει να αντιμετωπιστεί όταν ο A.E.P υλοποιηθεί σε I.D.S.

4.7.2 Η επιλογή του RA_Threshold σε συνάρτηση με το Threshold του A.E.P

Όπως προαναφέραμε η επιλογή του RA_Threshold θα γίνεται πάντα σε συνάρτηση με το Threshold που θα επιλέγεται για κάθε πόρτα, δηλαδή για κάθε Client-Server επικοινωνία. Αυτή η επιλογή γίνεται σαν αποτέλεσμα της παρατήρησης ότι όταν υπάρχει μεγάλο sledge πολύ πιθανό να εμφανίζεται πολλές φορές η RA.

Ο λόγος που υπάρχει ένα sledge σε ένα B.O.E όπως εξηγείται αναλυτικά στο Α μέρος της πτυχιακής, είναι για να αυξηθεί η πιθανότητα επιτύχιας του. Όσο μεγαλύτερο είναι το Buffer τόσο δυσκολότερο είναι να βρεθεί η σωστή RET, άρα και τόσο μεγαλύτερο Sledge χρειάζεται. Ακόμα από εμπειρία γνωρίζουμε ότι συνήθως όταν το Buffer είναι μεγάλο ο Blackhat προτιμά να βάλει μέσα στο Exploit πολλές φορές την Return Address για να εξασφαλίσει ότι η RET θα **καταπατήσει(override)** την RA του προγράμματος, ώστε να αυξήσει τις πιθανότητες επιτυχίας του B.O.E. Αντίθετα στην περίπτωση που το Buffer είναι μικρό τότε θα πρέπει να είναι πολύ πιο προσεχτικός γιατί αναγκάζεται να βάλει μικρότερο Sledge αλλά και μικρότερο αριθμό από επαναλαμβανόμενα RA. Από όλα αυτά **συμπεραίνουμε ότι όταν υπάρχει μεγάλο Sledge τότε η ποσότητα των RA μάλλον θα είναι μεγάλη**.

Εκτός όμως από τις παραπάνω περιπτώσεις υπάρχουν και πολλές άλλες για τις οποίες δεν ισχύει αυτή η αναλογία, είτε εκούσια, για να μπορέσει ο Blackhat να αποφύγει πιθανούς μηχανισμούς εντοπισμού, είτε ακούσια, λόγω της ιδιόμορφης δομής που πρέπει να έχει το B.O.E ώστε να πετύχει. Για να μπορέσουν να καλυφθούν όσο είναι δυνατόν όλες αυτές οι περιπτώσεις θα χρειαστεί να επιλεγεί ένα Threshold για τον εντοπισμο των RA αρκετές φορές υποπολλαπλάσιο του Threshold που χρησιμποιείται για τον εντοπισμό του Sledge.

ΣΥΜΠΕΡΑΣΜΑ
Για να έχουμε ένα επιτυχημένο μηχανισμό Validation χρειάζεται ένα Threshold που να πιάνει όσο το δυνατόν περισσότερες παραλλαγές της ύπαρξής του RA μέσα σε ένα B.O.E. Για να γίνει αυτό με βάση την εμπειρία μας θα πρέπει να το RA_Threshold να είναι σε συνάρτηση με το Threshold για το Sledge. Η συνάρτηση αυτή με βάση τα παραπάνω θα πρέπει να είναι τέτοια που θα μειώνει σημαντικά το RA_Threshold σε σχέση με το Threshold. Αυτό μπορεί να γίνεται με μία απλή διαίρεση:

RA_Threshold**==**Threshold **/ x** όπου **x>=3**

4.8 Η διαφορά υλοποίησης από Apache module σε NIDS

4.8.1 Γενικά

Στην αρχή αυτού του κεφαλαίου αναφέρθηκε ότι στα πλαίσια αυτής της πτυχιακής θα εφαρμοστεί ο αλγόριθμος A.E.P σε συνδυασμό με τον V.F.R.A σαν module για ένα NIDS. Το N.I.D.S αυτό είναι Open Source και λέγεται Snort. Λεπτομέρειες για αυτό και για την υλοποίηση του αλγορίθμου A.E.P θα δούμε στο κεφάλαιο 5 της πτυχιακής. Εδώ θα αναφέρουμε όμως μερικές βασικές δυσκολίες που θα αντιμετωπιστούν κατά την υλοποίηση.

4.8.2 Η Βασική δυσκολία κατά την υλοποίηση του A.E.P σε N.I.D.S

Η **μεγαλύτερη δυσκολία** είναι **η επιλογή του Threshold**. Στην περίπτωση του Apache Module επιλέχθηκε μια μόνο τιμή γιατί το πρόβλημα των B.O.E αντιμετωπίζεται για μία μόνο υπηρεσία και για μια συγκεκριμένη υλοποίηση καθώς επίσης για μία μόνο πλατφόρμα (Linux-

Intel). Στην περίπτωση του NIDS όμως το πρόβλημα είναι ότι μέσα από ένα δίκτυο που αυτό παρακολουθεί, μπορούν να περάσουν B.O.E προς **οποιαδήποτε υπηρεσία** και **οποιαδήποτε πλατφόρμα** κάτι που καθιστά δύσκολο για το NIDS να εντοπίζει αποτελεσματικά όλες τις περιπτώσεις B.O.E. χρησιμοποιώντας ένα Threshold για όλες τις υπηρεσίες.

Στο πείραμα που έκαναν οι εμπνευστές του αλγορίθμου παρατηρούμε ότι δοκίμασαν B.O.E που απευθύνονταν σε διαφορετικές υπηρεσίας όπως το FTP. Από αυτό διαπιστώθηκε ότι και σε αυτές τις περιπτώσεις με το ίδιο Threshold μπόρεσε ο αλγόριθμος να εντοπίσει το Exploit. Αυτό είναι ένα ιδιαίτερα ευχάριστο αποτέλεσμα γιατί μας δείχνει ότι το **ίδιο Threshold μπορεί να είναι πολύ αποτελεσματικό και σε άλλες περιπτώσεις**. Επιπλέον βασιζόμαστε στην υπόθεση που και ο ίδιος ο AEP στηρίχτηκε από την αρχή δηλαδή ότι **τα sledge προκαλούν σε ένα πακέτο ιδιαίτερα μεγάλα I.C. με μεγάλο E.L**. Με βάση τις δύο τελευταίες παρατηρήσεις θα χρησιμοποιούμε την τιμή **30 σαν Threshold για όλες τις γνωστές υπηρεσίες δηλαδή για όλες τις UDP και TCP πόρτες** αφού σε κάθε μια υπηρεσία αντιστοιχεί τουλάχιστόν μία πόρτα.

ΣΗΜΕΙΩΣΗ
Επειδή παρατηρήσαμε ότι το Threshold 30 είναι μια ικανοποιητική και για άλλες περιπτώσεις Client-Server επικοινωνίας θα χρησιμοποιήσουμε αρχικά κατά την υλοποίηση του AEP σε IDS την ίδια τιμή. Η τιμή αυτή όπως φαίνεται είναι αρκετά ικανοποιητική τουλάχιστόν για να μην έχουμε False Negative. Για να εξασφαλιστεί ότι δεν θα υπάρχουν πολλά False Positive θα χρησιμοποιηθεί μεγαλύτερο Threshold για πόρτες που δεν χρησιμοποιούνται από γνωστές υπηρεσίες. Για να μπορέσει όμως ο μηχανισμός να δουλέψει όσο το δυνατόν καλύτερα **χρειάζονται πειραματικά δεδομένα** αντίστοιχα με αυτά που παρουσιάστηκαν παραπάνω αλλά αυτή τη φορά για την συνολική κίνηση ενός δικτύου συνδεμένο με το Internet.

4.8.3 Άλλες Δυσκολίες

Εκτός από το πρόβλημα του Threshold που όπως είπαμε είναι κάτι που για να λυθεί χρειάζεται πειραματική μελέτη υπάρχεί και ένα σύνολο άλλα προβλήματα. Τα προβλήματα αυτά προκύπτουν συνήθως από την ιδιαίτερη συμπεριφορά που έχουν κάποια Services. Σε μία συνηθισμένη συμπεριφορά για να εκτελεστεί το αίτημα του Client συνήθως υπάρχει ένας μηχανισμός decoding που θα διαχωρίσει τις εντολές από τα δεδομένα ώστε μετά να κλιθούν οι ανάλογες συναρτήσεις χειρισμού. Για παράδειγμα στο telnet πρωτόκολλο το μόνο που θα γίνει είναι ο διαχωρισμός των εντολών από τα δεδομένα. Σε πολλές περιπτώσεις όμως τα ίδια τα δεδομένα χρειάζονται και αυτά κάποια μορφή **αποκωδικοποίησης** ή **κανόνικοποίησης.**

Οι περιπτώσεις των πακέτων αυτών είναι τα **HTTP πακέτα** και τα **SMTP πακέτα**. Ακόμα και τα SSL πακέτα πρέπει πρώτα να αποκωδικοποιηθούν για να γίνει έλεγχος για B.O.E. Το τελευταίο όμως δεν είναι δυνατόν να συμβεί σε *πραγματικό χρόνο(Real Time)*.

Για παράδειγμα οι χαρακτήρες από ASCII μπορούν πρώτα να κωδικοποιηθούν σε Unicode πριν αποσταλούν στον Server όταν αυτός είναι ο IIS. Άλλο παράδειγμα είναι ότι μπορεί να χρησιμοποιηθεί η πισωκάθετος (\) αντί για την κάθετο (/) σαν χαρακτήρας διαχωρισμού σε ένα URI. Όλα αυτά πριν ελεγχθούν από τον AEP για πιθανό BOE θα πρέπει πρώτα να **αποκωδικοποιηθούν** ή να **κανονικοποιηθούν.** Ευτυχώς για εμάς αυτό θα το κάνει ο HTTP_decode που είναι ένας Snort preprocessor. Είναι δηλαδή ένα module του snort που είναι υπεύθυνο να **αποκωδικοποιεί** και να **κανονικοποιεί** τα HTTP πακέτα με σκοπό όλα τα υπόλοιπα μέρη του Snort να παίρνουν **την standard URI μορφή** των πακέτων γιά να τα επεξεργαστούν και να ελέγξουν αν αυτά τα πακέτα είναι μέρος μίας επίθεσης.

Εκτός από το HTTP υπάρχει και η περίπτωση του SMTP. Όταν πρωτοσχεδιάστικε το πρωτόκολλο του e-mail η ανάγκη που είχε αρχικά αποφασιστεί να καλύπτει ήταν η αποστολή **απλών κειμένων(plain text)** όμως αργότερα προέκυψε και η ανάγκη να μεταφερθούν και άλλες πληροφορίες όπως εικόνα ήχος μέχρι και Video. Για να μην χρειαστεί να αλλάξει ριζικά το πρωτόκολλο αποφασιστηκέ να δημιουργηθεί η κωδικοποίηση του MIME. Με αυτό τον

τρόπο όλα μπορούν να διακινούνται μέσω e-mail σε μορφή κειμένου αρκεί να αναφέρεται στην κεφαλίδα του πακέτου(Header) ότι το περιεχόμενο είναι εικόνα, ήχος ή κείμενο για να μπορεί να γίνει από τον παραλήπτη η σχετική αποκωδικοποίηση. Η βασικότερη κωδικοποίηση που χρησιμοποιεί είναι το base64. Δυστυχώς το Snort δεν διαθέτει έναν τέτοιο μηχανισμό αποκωδικοποίησης για base64, άρα θα χρειαστεί να προστεθεί και να εκτελείται πριν από την εκτέλεση του ΑΕΡ αν διαπιστωθεί ότι χρειάζεται.

Αυτά αλλά και ένα σύνολο από άλλα παραπλήσια προβλήματα πρέπει να προσεχθούν ώστε το module του NIDS Snort που θα φτιαχτεί να έχει όσο το δυνατόν καλύτερη απόδοση. Η απόδοση πρέπει να περιλαμβάνει αφενός την αναγνώριση με ακρίβεια όλων των Β.Ο.Ε αφετέρου αυτό να μπορεί να γίνεται χωρίς κατασπατάληση επεξεργαστικής ισχύς ώστε να εφαρμόζεται σε **πραγματικό χρόνο(Real Time)**

4.9 Σύνοψη – Παρατηρήσεις

Στο κεφάλαιο αυτό παρουσιάζεται αναλυτικά η μέθοδος Α.Ε.Ρ για τον εντοπισμό των Β.Ο.Ε καθώς και οι προσθήκες που έγιναν όπως ο αλγόριθμος V.F.R.A. Η παρουσίαση αυτή έγινε κυρίως για να είναι κατανοητό πως δουλεύει ο **Snort Preprocessor** που θα παρουσιαστεί στο Κεφάλαιο 5 της πτυχιακής.

Η βασική ιδέα είναι να χρησιμοποιηθεί ο αλγόριθμος Α.Ε.Ρ σαν κύριος μηχανισμός για να εντοπίζονται επιθέσεις ΒΟΕ μέσω ενός NIDS. Με αυτό τον τρόπο **γενικεύεται η χρήση του αλγόριθμου αυτού .**

Όπως αναλύεται παραπάνω υπάρχουν διάφορες δυσκολίες που πρέπει να αντιμετωπιστούν κατά την εφαρμογή αυτή της μεθόδου που προκύπτουν μόνο όταν αυτό θα χρησιμοποιηθεί σαν Preprocessor του Snort και όχι σαν Apache module. Ένα ακόμα πρόβλημα που θα πρέπει να αντιμετωπιστεί είναι οι περιπτώσεις των Heap Based Buffer Overflows.

4.9.1 Η εύνοια των συμπτώσεων και τα Heap Bases ΒΟΕ

Σε ολόκληρο το κεφάλαιο παρουσιάσθηκε η λύση του Α.Ε.Ρ με αναφορές στο **Stack Based Based Buffer Overflow** και όχι στο **Heap Based Buffer Overflow**. Αυτό έγινε σκόπιμα για να μπορέσει να γίνει κατανοητός ο ΑΕΡ χωρίς να χρειάζεται να απασχοληθεί ο αναγνώστης με της ιδιαιτερότητες των **Heap Based ΒΟΕ**. Τώρα που έχει γίνει κατανοητό το πώς λειτουργεί ο ΑΕΡ θα παρουσιαστεί ποια κατά την εκτίμηση του συγγραφέα αυτής της πτυχιακής θα είναι η συμπεριφορά του αλγορίθμου όταν θα έχει να αντιμετωπίσει Heap Based ΒΟΕ.

Τα Heap Based ΒΟΕ όπως περιγράφεται αναλυτικά στο **κεφάλαιο 2** χωρίζονται σε δύο μεγάλες κατηγορίες. Στην πρώτη κατηγορία ανήκουν τα Heap Β.Ο.Ε που απλώς αντικαθιστούν το όνομα ενός αρχείου ή ένα Password. Η δεύτερη κατηγορία που είναι και η ποιό συνηθισμένη είναι αυτή που αντί για την RET θα πρέπει να αντικαθιστά ένα **Function Pointer** όπως για παράδειγμα το **openSSL remote exploit**.

Από τις περιπτώσεις αυτές η δεύτερη είναι σχεδόν ίδια με αυτή του συνηθισμένου **Stack Based ΒΟΕ** συνεπώς αναμένεται ο ΑΕΡ να μπορεί να **εντοπίσει(Detect)** τις περιπτώσεις αυτές των **Heap Based ΒΟΕ**. Όσο αφορά τον V.F.R.A αναμένεται ότι και αυτός θα μπορέσει να αποδώσει αν ο Pointer μέσα σε ένα Β.Ο.Ε String εμφανίζεται αρκετές φορές. Αυτό θα συμβεί επειδή ήδη έχει προβλεφθεί ότι εκτός από αναφορές προς την στοίβα θα πρέπει να ελέγχονται και αναφορές προς την περιοχή που φορτώνονται τα **DLL** ή άλλες **δυναμικά συνδεόμενες βιβλιοθήκες.**

Το πρόβλημα είναι πώς θα μπορούσαν να αντιμετωπισθούν τα Heap Based ΒΟΕ της πρώτης κατηγορίας που αναφέρεται παραπάνω καθώς και οι περιπτώσεις των Heap ή Stack Based ΒΟΕ. Το πρόβλημα και στις τρεις περιπτώσεις είναι ότι δεν **υπάρχει Sledge** ώστε να **αυξηθεί ο βαθμός εκτελεσιμότητας** των **ΒΟΕ string** και τελικά να μπορέσει ο ΑΕΡ να τα εντοπίσει. Το πρόβλημα αυτή την φορά το λύνει η **«σύμπτωση»** που περιγράφετε στην **παράγραφο 4.4.4.2**.

Η **«σύμπτωση»** που είναι το **μεγαλύτερο πρόβλημα που έχει να αντιμετωπίσει ο αλγόριθμος** ΑΕΡ για τις τρεις **παραπάνω περιπτώσεις, είναι η λύση**. Αυτό συμβαίνει γιατί

συνήθως αν και δεν υπάρχει ανάγκη για Sledge σε μερικές περιπτώσεις, σε αυτές τις ίδιες περιπτώσεις χρειάζεται να γεμιστεί το Buffer με ένα σύνολο από χαρακτήρες που ο μόνος ρόλος τους θα είναι να προκαλέσουν υπερχείλιση. Συνήθως όμως **επιλέγεται ένας μόνο τυχαίος χαρακτήρας** που θα παίξει το ρόλο του **Filler(γεμιστεί)**. Από την άλλή μεριά η έννοια της **«σύμπτωσης»** μας λέει ότι αν ένας χαρακτήρας επαναλαμβάνεται πολλές φορές και αυτό **αντιστοιχεί σε κάποια εντολή του επεξεργαστή τότε ο AEP θα θεωρήσει ότι αυτή η ακολουθία των χαρακτήρων είναι Sledge ακόμα και αν δεν χρησιμοποιείται σαν Sledge**. Μία τέτοια περίπτωση είναι και το **openSSL remote exploit** που περιγράφεται **στο Παράρτημα A1**.

Από τα παραπάνω φαίνεται ότι η «σύμπτωση» είναι επιθυμητή όμως πρέπει να γίνουν κατανοητά δύο πράγματα. Η «σύμπτωση» συνήθως είναι πρόβλημα γιατί εξαιτίας της δεν μπορούν πάντα εύκολα να διαχωριστούν τα BOE από τα **υπόλοιπα πακέτα.** Το **δεύτερο είναι ότι θεωρείται στην πράξη πολύ πιθανό να υπάρχουν ή να υπάρξουν B.O.E ειδικά Heap Based που δεν θα μπορούν να εντοπιστούν αν αυτά δεν χρειάζονται sledge**. Το τελευταίο μπορεί να γίνει αν **αντί για την ακολουθία ενός μόνο χαρακτήρα** χρησιμοποιηθεί **μια ακολουθία από διαφορετικούς χαρακτήρες μεταξύ τους που ο συνδυασμός τους δεν αντιστοιχεί σε εντολές** του επεξεργαστή.