Κεφάλαιο 5
Buffer Overflow Detection Preprocessor

5.1 Γενικά

Σε αυτό το κεφαλαίο παρουσιάζεται ο Preprocessor που αναπτύχθηκε στα πλαίσια αυτής της πτυχιακής και ονομάζεται "**BufferOverflowDetector"**. Το κεφαλαίο αυτό στην πράξη είναι το αποτέλεσμα της συνολικής εργασίας που έγινε πάνω στο πώς γίνονται οι επιθέσεις των Blackhat που βασίζονται στα Buffer Overflow αλλά και πώς αυτές οι επιθέσεις(Buffer Overflow Exploit) μπορούν να **εντοπιστούν αποτελεσματικά(Accurate Detection)**. Η κατασταλτική μέθοδος που χρησιμοποιείται σε αυτό τον Preprocessor είναι φυσικό αποτέλεσμα της προσπάθειας να περιοριστούν τα **Buffer Overflow Exploit** μέχρι να γίνει η πλήρης μετάβαση σε συστήματα που δεν υποφέρουν από τέτοιες αδυναμίες. Όπως φαίνεται στο **κεφαλαίο 3** μέχρι σήμερα δεν υπάρχει κάποια λύση που να αντιμετωπίζει καθολικά το πρόβλημα ενώ από ότι φαίνεται ο μόνος τρόπος για να λυθεί είναι να ξανασχεδιαστούν οι Compiler τις C/C++ ώστε να ενσωματώνουν πλήρες Boundary Checking και όχι να υιοθετούνται λύσεις όπως το **StackGuard**.

Για να γίνει κατανοητός ο πηγαίος κώδικας του BufferOverflowDetector Preprocessor προηγουμένως θα επεξηγηθούν οι σημαντικότερες συναρτήσεις που αποτελείται ο Preprocessor αλλα θα γίνει και μια **εισαγωγή στο Snort 2.0**.

Πριν προχωρήσουμε στην περαιτέρω ανάλυση του Buffer Overflow Detector Preprocessor θα πρέπει να τονιστεί κάτι πολύ σημαντικό. Η μέθοδος που χρησιμοποιεί ο **BufferOverflowDetector Preprocessor** έχει σαν κύριο στόχο να εντοπίζει **Buffer Overflow Exploits** που **δεν είναι γνωστά**. Με αυτή την πρόταση πρέπει να γίνει κατανοητό ότι η μέθοδος αυτή είναι μέθοδος **Anomaly Based Detection**. Έτσι λοιπόν αυτή η μέθοδος που χρησιμοποιείται από τον Preprocessor, όπως και όλες οι μέθοδοι που ανήκουν στην κατηγορία των μεθόδων Anomaly Based Detection, είναι πιθανό να κάνει λάθος. Το λάθος αυτό έστω και μικρό αν είναι δεν μπορεί να αγνοηθεί και πάντα χρειάζεται ο ανθρώπινος παράγοντας **που θα επιβεβαιώσει την εγκυρότητα της απόφασης** του BufferOveflowDetector για **ένα πακέτο**.

Τέλος πρέπει να επισημανθεί ότι όπως σε όλες τις μεθόδους που η λύση τους βασίζεται στην στατιστική για να **μετρηθεί το σφάλμα** που μπορεί να κάνει ο αλγόριθμος πρέπει να γίνουν αρκετές πειραματικές μετρήσεις. Τέτοιες μετρήσεις έχουν ήδη γίνει για την μέθοδο που χρησιμοποιεί ο Preprocessor όπως φαίνεται στο **κεφάλαιο 4** όμως η αποτελεσματικότητα της έχει φανεί μόνο για ένα περιορισμένο αριθμό περιπτώσεων. Οι μετρήσεις αυτές **ξεφεύγουν από τα όρια αυτής τις πτυχιακής**.

5.2 Εισαγωγή στο Snort 2.0

5.2.1 Γενικά

Όπως αναφέρθηκε στο **κεφάλαιο 3** ένας από τους τρόπους αντιμετώπισης των επιθέσεών που γίνονται μέσο δικτύου και στηρίζονται στις **Buffer Overflow Vulnerabilities** είναι η χρήση των NIDS. Με τα NIDS καθίσταται δυνατό να μπορεί να ειδοποιηθεί ο διαχειριστής του δικτύου για τέτοιου είδους επιθέσεις αλλά και να εμποδίζει αυτές τις επιθέσεις, αν το επιθυμεί, κόβοντας τα ύποπτα πακέτα πριν φτάσουν στον προορισμό τους και πετύχουν τον σκοπό τους.

Σε αυτή την παράγραφο θα παρουσιατεί το Snort2.0 πού είναι ένα από τα πλέων γνωστότερα open source NIDS. Συγκεκριμένα θα περιγραφεί η αρχή λειτουργίας του και οι μεθόδοι που κάνει εντοπισμό. Ακόμα θα περιγραφεί ένα μέρος του μηχανισμού σε επίπεδο development ώστε να εξηγηθεί το πώς μπορεί κάποιος να φτιάξει ένα preprocessor σαν plug-in του snort.

5.2.2 Γενική περιγραφή του Snort2.0

Το Snort είναι ένα Open Source και lightweight NIDS που μέχρι την έκδοση 1.9.1 το χρησιμοποιούσαν σε δίκτυα μικρού σχετικά μεγέθους και με μικρό σχετικά bandwidth μέχρι

100Mbps. Όμως από την έκδοση 2.0 και μετά άλλαξε ριζικά ο μηχανισμός εντοπισμού (Detection engine) με την νέα "Hi-performance Multi-Rule Inspection engine" και έτσι το snort μπορεί να χρησιμοποιηθεί και σε δίκτυα με ***Gigabit*** Bandwidth.Ο δημιουργός του είναι ο *Martin Roesch* και ο κώδικας του είναι γραμμένος σε C.

Το Snort εκτός από την λειτουργία του σαν NIDS μπορεί να δουλέψει και σαν ένας απλός sniffer ή σαν ένας sniffer που καταγράφει(logging) τα πακέτα που λαμβάνει σε log αρχεία σε μορφή απλού κειμένου ASCII. Έχει λοιπόν τρία mode λειτουργίας που περιγράφονται παρακάτω είναι :

1. **Sniffer** mode.
2. **Packet logger** mode.
3. **NIDS** mode.

5.2.2.1 Sniffer Mode

Σε αυτό το mode λειτουργίας το Snort έχει την ικανότητα να διαβάζει τα πακέτα που περνάνε από το δίκτυο, να τα αποκωδικοποιεί και να τα εμφανίζει στην οθόνη σε φιλική μορφή για τον χρήστη.

Ο χρήστης με διάφορα BPF (Berkley Packet Filter) φίλτρα που μπορεί να χρησιμοποιήσει, έχει την δυνατότητα να ορίσει το είδος των πακέτων που θα εμφανίζονται όσο αναφορά το πρωτόκολλο, τον αποστολέα, τον παραλήπτη και διάφορα άλλα χαρακτηριστικά ενός πακέτου. Για παράδειγμα αν γράψει την λέξη κλειδί **icmp** στο snort τότε αυτό θα δείχνει μόνο τα πακέτα που είναι τύπου ICMP. Η λειτουργία αυτή του Snort είναι παρόμοια με αυτή του γνωστού εργαλείου tcpdump, το οποίο διατίθεται κυρίως με τα περισσότερα *λειτουργικά συστήματα* της οικογένειας του **Unix**.

5.2.2.2 Packet Logger Mode

Σε αυτό το mode λειτουργίας το Snort αποθηκεύει στο δίσκο τα πακέτα που διαβάζει από το δίκτυο, αντί απλά να τα εμφανίζει στην οθόνη. Η διαδικασία αυτή είναι αρκετά σημαντική στην περίπτωση που απαιτείται τα πακέτα αυτά να εξεταστούν με λεπτομέρεια σε επόμενο στάδιο. Το Snort μπορεί να αποθηκεύσει τα πακέτα αυτά σε διάφορα formats, ανάλογα με τις ανάγκες του χρήστη. Για παράδειγμα μπορεί να αποθηκεύσει τα πακέτα σε binary μορφή (tcpdump format), με την οποία μπορούν να χρησιμοποιηθούν σαν είσοδο σε διάφορα άλλα προγράμματα ανάλυσης πακέτων και πρωτοκόλλων, σε ASCII μορφή ώστε να είναι δυνατή η ανάγνωσή τους, σε XML μορφή ή και να οργανωθούν σε βάσεις δεδομένων.

Είναι σημαντικό να σημειωθεί εδώ ότι αυτό το mode μπορεί να λειτουργεί παράλληλα με το *sniffer mode* ή το *NIDS mode* και δεν λειτουργεί υποχρεωτικά ανεξάρτητα από τα αλλά mode.

5.2.2.3 NIDS Mode

Αυτή είναι η κύρια λειτουργία του Snort. Όπως αναφέρθηκε προηγουμένως, το Snort είναι ένα IDS το οποίο ενεργεί σε επίπεδο δικτύου, δηλαδή τα γεγονότα που παρακολουθεί και εξετάζει για την εμφάνιση μίας πιθανής επίθεσης, αφορούν την δραστηριότητα που παρατηρείται σε ένα δίκτυο. Το Snort έχει την ικανότητα να ανιχνεύει ένα μεγάλο φάσμα από γνωστές δικτυακές επιθέσεις, όπως *portscans, buffer overflows, OS fingerprints* και πολλά άλλα.

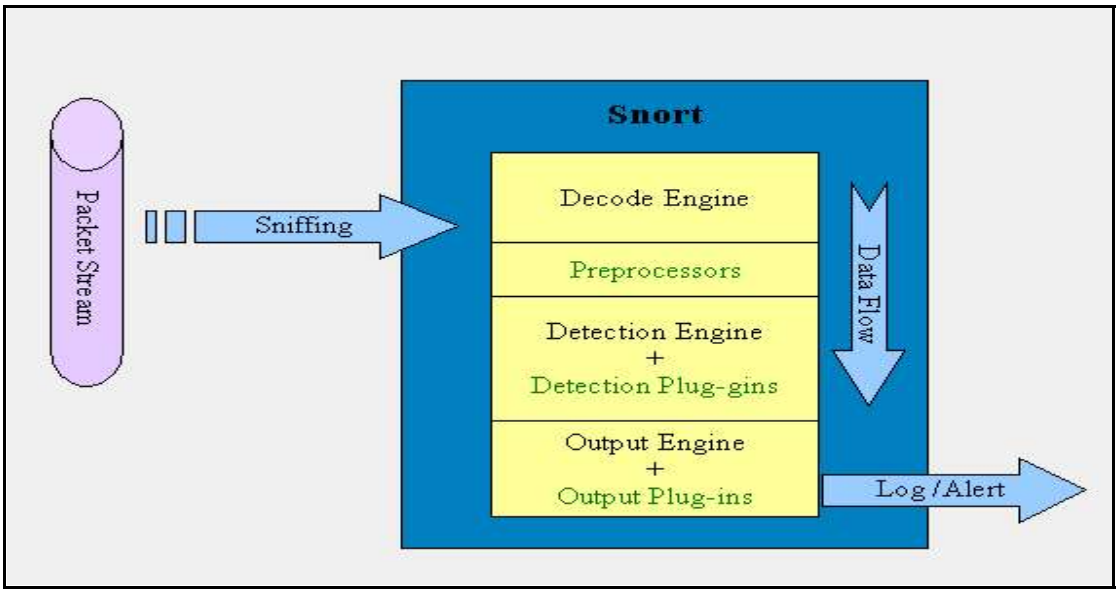
Η τεχνική που χρησιμοποιεί το Snort για την διαδικασία αυτή είναι κατά κύριο λόγο η *Misuse Detection* με την χρήση των *Signatures* ενός βλαβερού(malicious) πακέτου. Το snort όμως ειδικά μετά την έκδοση 2.0 συνδυάζει την λειτουργία της ανάλυσης των γεγονότων με κάποιες από τις μεθόδους του ***Protocol Anomaly Detection*** (http://www.snort.org/documentation) και του ***Anomaly Detection*** για την ανίχνευση πιθανών επιθέσεων. Οι μηχανισμοί αυτοί υλοποιούνται κατά κύριο λόγο από τους preprocessors που εξηγούνται αναλυτικά παρακάτω αλλά και από το νέο μηχανισμό του snort 2.0 να συντάσσει τα rules.

5.2.3 Η Μηχανή του snort2.0

Το Snort αποτελείται από τέσσερα υποσυστήματα λειτουργίας. Κάθε πακέτο που επεξεργάζεται το Snort, θα περάσει από κάθε ένα από αυτά τα υποσυστήματα :

- Packets capturing και decoding engine.
- Rules parsing και detection engine που αντικαταστάθηκε με την Hi-performance Multi-Rule Inspection engine.
- Logging ή Output engine.
- Detection plugins, Output plugins and **Preprocessors** handling engine.

Το σχημα που ακολουθεί παρουσιάζει την δομή του Snort, όσο αναφορά τα υποσυστήματα από τα οποία αποτελείται και αναπαριστά την διαδρομή που ακολουθεί κάθε πακέτο κατά την επεξεργασία του από το Snort.



Απο όλοκληρο τον μηχανισμό του Snort αυτό που μας ενδιαφέρει στα πλαίσια αυτής της πτυχιακής είναι οι Preprocessors και για αυτό παρακάτω θα γίνει μία πολύ σύντομη περιγραφή των **Snort 2.0 Preprocessors**. Περισσότερες πληροφορίες για το Snort και για τους Preprocessor μπορείεται να βρείται στο **Technical Report "Snort 2&Snort Preprorcessors"** στο βιβλείο **"Snort 2.0"** και στο URL http://www.snort.org/.

5.2.4 Τι είναι και τι προσφέρει ένας preprocessor

Η δυνατότητα NIDS όπως το Snort2.0 να εφαρμόσουν Rule/Signature pattern matching είναι πολύ δημοφιλής γιατί είναι ***εξαιρετικά γρήγορος*** τρόπος για να παρακολουθείται ένα δίκτυο με ***μεγάλη κίνηση και bandwidth*** και είναι σίγουρο ότι θα ελέγχθούν όλα τα πακέτα που πιθανόν είναι βλαβερά(malicious). Εδώ όμως προκύπτει το πρόβλημα ότι τα Rules που γράφονται δεν μπορεί να είναι πολύ γενικά γιατί τότε θα προκαλούν πολλά **false positive** ούτε όμως πολύ συγκεκριμένα γιατί τότε θα προκαλούν πολλά **false negative.** Αυτή είναι μια εν γένη αδυναμία των NIDS και η αδυναμία αυτή στο Snort περιορίζεται με τους **Preprocessors.** Ένας preprocessor όπως περιγράψαμε και στο προηγούμενο κεφαλαίο δίνει την δυνατότητα να κρατά και άλλες πληροφορίες όπως τύπου anomaly based detection και normalization που δίνουν την δυνατότητα στο Snort να κάνει πιο αποτελεσματικά την δουλεία του. Για παράδειγμα αν σε ένα rule περιγράφεται ότι η παρουσία της εντολής *cmd.exe* είναι πιθανών *buffer overflow based Attack* τότε δεν θα εμφανιστεί το σχετικό alert αν κάποιος preprocessor προηγουμένως διαπιστώσει ότι υπάρχει κάποια **ανωμαλία στο πακέτο, με βάση το πρωτόκολλο, στο οποίο περιέχεται αυτή η εντολή.**

Ένας **Preprocessor** είναι ένα αυτόνομο πρόγραμμα που συνδέεται σαν module(άρθρομα) με τον υπόλοιπο μηχανισμό του snort. Ένας preprocessor μπορεί να λειτουργεί τελείως

αυτόνομα ή να συνεργάζεται ή να εξαρτάτε από άλλους preprocessors. Η δυνατότητα να μπορεί να μπαίνει στο snort ένα πρόγραμμα στην μορφή ενός plug-in και να ενεργοποιείται ή να απενεργοποιείται κατά βούληση του διαχειριστή του δικτύου δεν δίνει μόνο μεγάλη δυνατότητα επέκτασης του snort αλλά και την δυνατότητα να προσαρμόζεται στις ανάγκες οποιουδήποτε δικτύου. Όπως θα δούμε παρακάτω στο αρχείο **snort.conf** μπορούμε να επιλέξουμε ποίοι από τους preprocessor θέλουμε να ενεργοποιήσουμε και ακόμα ποίες από τις υπολειτουργίες του κάθε preprocessor θέλουμε να ενεργοποιήσουμε και ποιές όχι. Η γενική σύνταξη για να ενεργοποιηθεί ένας preprocessor αφού προηγούμενος έχει γίνει compiled με το υπόλοιπο snort είναι:

```
Preprocessor <preprocessor> : <options>
```

- Preprocessor : είναι η λέξη κλειδί που θα ενεργοποιήσει κάποιον preprocessor
- <preprocessor> : είναι το όνομα του preprocessor που θέλουμε να ενεργοποιηθεί για παράδειγμα frag2 ή stream4.
- <options> : είναι ένα σύνολο από επιλογές που μπορεί να γίνουν σε κάποιον preprocessor. Για παράδειγμα αν θέλουμε ένας preprocessor να ενεργοποιείται για ένα συγκεκριμένο σύνολο από πόρτες προορισμού(Destination Ports) τότε βάζουμε μία λίστα με αυτές τις πόρτες σαν <options>.

NOTE
Όπως είπαμε και σε προηγούμενα κεφάλαια η τεχνολογία των NIDS δεν είναι τέλεια και παρουσιάζει αρκετές αδυναμίες στον εντοπισμό των βλαβερών πακέτων(malicious). Τα πακέτα αυτά**, που αδυνατεί ένα NIDS να πάρει την σωστή απόφαση,** τα ξεχωρίζουμε σε δύο κατηγορίες.

- Με το όρο **False Positive** χαρακτηρίζονται τα πακέτα που ενώ **ΔΕΝ** είναι βλαβερά(malicious) ο μηχανισμός του IDS τα χαρακτηρίζει σαν βλαβερά και προκαλεί σχετικό alert. Για παράδειγμα αν μέσα σε ένα πακέτο εμφανιστούν συμπτωματικά, ή λόγο κακής λειτουργίας του δίκτυου, *signatures* τότε θα εμφανιστεί σχετικό alert προκαλώντας «ψευδή συναγερμό» ενώ δεν θα έπρεπε. Άλλο παράδειγμα false positive παρουσιάζεται σε δίκτυα που έχουν *windows98* λειτουργικό σύστημα. Σε αυτά τα συστήματα έχει παρατηρηθεί ότι σε πολλά TCP transaction στα πακέτα στέλνονται περισσότερα δεδομένα ανά πακέτο από το εκάστοτε **TCP stream window** που προηγμένος έχει συμφωνηθεί από το TCP πρωτόκολλο. Αυτό φυσικά θα προκαλέσει alert αλλά είναι ψευδές γιατί οφείλεται στην ιδιόμορφή λειτουργία του ίδιου του λειτουργικού συστήματος.

- Με το όρο **False Negative** χαρακτηρίζονται τα πακέτα που ενώ θα έπρεπε για αυτά το IDS να προκαλέσει κάποιο alert δεν αντιδρά καθόλου σε αυτό. Για παράδειγμα αν κάποιος **κακόβουλος χρήστης(blackhat)** παράγει μια παραλλαγή(*βλέπε πολυμορφικά buffer overflow exploits*) ενός **buffer overflow based exploit** που για αυτό δεν υπάρχει σχετικό rule ή κάποιος άλλος μηχανισμός, όπως ένας preprocessor, τότε το πιθανότερο είναι το exploit αυτό να περάσει απαρατήρητο και χαρακτηρίζεται σαν *false negative*.

5.3 Οι συναρτήσεις του BufferOverflowDetector Preprocessor

Preprocessor Έχει ένα πλήθος συναρτήσεων που χρησιμοποιεί για να εκτελέσει το αλγόριθμο **AEP** αλλά και ένα σύνολο από άλλες λειτουργίες που είναι απαραίτητες για την συνεργασία του με το Snort ή για την προετοιμασία των συνθηκών ώστε ο αλγόριθμος να εκτελείται με επιτυχία.

Για να μπορέσει το Snort να ενσωματώσει τον Preprocessor στον υπόλοιπο κώδικα χρειάζεται να οριστούν 6 συναρτήσεις (λεπτομέριες στο Technical Report μου **"Snort 2.0 και Snort_Preprocessors"**). Για τον σκοπό αυτό στον BufferOverflowDetector υπάρχουν οι παρακάτω συναρτήσεις:

- void **SetupBuffOverDetector**();
- void **BuffOverDetectorInit**(u_char *);
- void **ParseBuffOverDetectorArgs**(char *);
- void **BuffOverDetectorFunction**(Packet *);
- void **PreprocRestartFunction**(int);
- void **PreprocCleanExitFunction**(int);

Οι συναρτήσεις αυτές αντιστοιχούν μία προς μία στις συναρτήσεις που περιγράφονται στο Technical Report "Προγραμματισμός(Configuration) και ανάπτυξη(Development) των Preprocessors". Από αυτές ο BufferOveflowDetector δεν χρησιμοποιεί τις **PreprocRestartFunction** και **PreprocCleanExitFunction** γιατί δεν χρειάζεται αλλά και δεν είναι υποχρεωτικό όπως αναφέρεται και στο Documentation του Snort. Οι υπόλοιπες έχουν ιδιαίτερη σημασία για τον Preprocessor.

Η **SetupBuffOverDetector**() εκτελεί τις απαραίτητες λειτουργίες για την ενσωμάτωση του Preprocessor στο Snort.

Η **BuffOverDetectorInit**() αρχικοποιεί τον Preprocessors και μία από τις λειτουργίες αρχικοποίησης που κάνει είναι να χτίσει το **δέντρο Trie που θα χρησιμοποιήσει** ο AEP για να ξεχωρίζει τους Opcode. Το δέντρο αυτό περιγράφεται αναλυτικά στο **κεφάλαιο 4.** Για να μπορέσει να χτιστεί αυτό το δέντρο καλείται η συνάρτηση **trieptr buildTrie(char * filename)** που ανήκει στην κατηγορία των συναρτήσεων του **BufferOveflowDetector** για αυτό το σκοπό.

Η **ParseBuffOverDetectorArgs**(char *) είναι η συνάρτηση που θα επεξεργαστεί την λίστα των παραμέτρων που υπάρχουν στο **snort.conf** για τον BufferOveflowDetector Preprocessor. Οι παράμετροι που προς το παρόν μπορεί να δεχτεί ο Preprocessor είναι ένα Threshold για κάθε TCP/UDP πόρτα. Τέλος μπορεί να επιλέγει για τις υπόλοιπες πόρτες να έχουν ή να μην έχουν καθόλου Threshold κατά συνέπεια να γίνεται ή να μην γίνεται έλεγχος για αυτές.

Η **BuffOverDetectorFunction**(Packet *) είναι η συνάρτηση που καλείται όταν το Snort θα παραδώσει το πακέτο στον Preprocessor. Με άλλα λόγια είναι η συνάρτηση που θα επεξεργαστεί το πακέτο όταν ο έρχεται η σειρά του **BufferOverflowDetector Preprocessor** να επεξεργαστεί ένα εισερχόμενο πακέτο. Στην πράξη αυτός είναι ο αλγόριθμος που εκτελεί ο Prepreccessor που με την σειρά του αυτός θα καλέσει τον αλγόριθμο AEP. Για αυτό παρακάτω θα παρουσιαστεί ένα **διάγραμμα ροής** για αυτό τον αλγόριθμο.

Οι συναρτήσεις που χρησιμοποιεί ο BufferOverflowDetector **για να χτίσει το δέντρο Trie** είναι οι παρακάτω.

- static void **checkEndOfString**(char * string);
- static char * **getEndOfTokenMultiple**(char * string, char * delimiters);
- static char * **getStartOfOpcode**(char * string);
- static char * **getStartOfNextOpcode**(char * string);
- static void **clear_node**(trieptr ptr);
- static trieptr **new_trie_node**(int nr_of_operands,int *ops, char * string, int opcode_length, unsigned char jump);
- trieptr **buildTrie**(char * filename);

- static **trieptr insertString**(trieptr trie, int number_of_opcode_bytes,int original_opcode_bytes,int * opcode, int number_of_args,int * args, char * string, unsigned char jump);

Συναρτήσεις σημαντικές για Debugging είναι οι παρακάτω που τυπώνουν τα περιεχόμενα του **δέντρου trie**.

- static void printTrieRecord(trieptr trie);
- static void printTrie(trieptr trie);
- static void myprint(trieptr trie);

Οι παρακάτω συναρτήσεις είναι στην πράξη κάποια εργαλεία που χρειάζεται ο Preprocessor για ειδικές περιπτώσεις:

- static bool decodeBase64(char *b64,char *ptext) : Χρησιμοποιείται για να **αποκωδικοποιεί πακέτα που έρχονται κωδικοποιημένα σε Base64 encoding** (δηλαδή τα e-mail).
- static int IsUberPacket(Packet *p): Χρησιμοποιείται για να αναγνωρίζει αν ένα πακέτο είναι “uber-paket” ή όχι. Λεπτομέρειες για τα πακέτα “uber” του Snort υπάρχουν στο Technical Report “Προγραμματισμός(Configuration) και ανάπτυξη(Development) των Preprocessors” στην περιγραφή του **Steam4 preprocessor.**

Οι παρακάτω συναρτίσεις στην πράξη υλοποιούν τον AEP μαζί με τον μηχανισμό αναγνώρισης των Opcodes.

- static trieptr searchtrie(trieptr ptr, unsigned char * code, int max_bytes);
- static int getAmountOfComsumedBytes(int type);
- static int getSkip(trieptr result);
- static long getPositionDifference(trieptr result, unsigned char * bytes, int counter);
- static int getMax(int a, int b);
- int analyzeRecord(trieptr r, unsigned char * bytes, int index, int length);
- static int get_ei(trieptr trie, unsigned char * bytes, int length, int startvalue,int bound, int startindex,char *passed_pos);
- static int test_for_opcodes(trieptr trie, unsigned char * bytes,int bytecode_size, int nr_of_instructions, int nr_of_tests);

Από αυτές οι σημαντικότερες είναι η **test_for_epodes()** και η **get_ei()** που υλοποιούν τον αλγόριθμο AEP του **κεφαλαίου 4**. Αν θέλετε να καταλάβετε το τι ακριβώς κάνουν δεν έχετε παρά να διαβάσετε τον αλγόριθμο AEP από το **κεφάλαιο 4.**

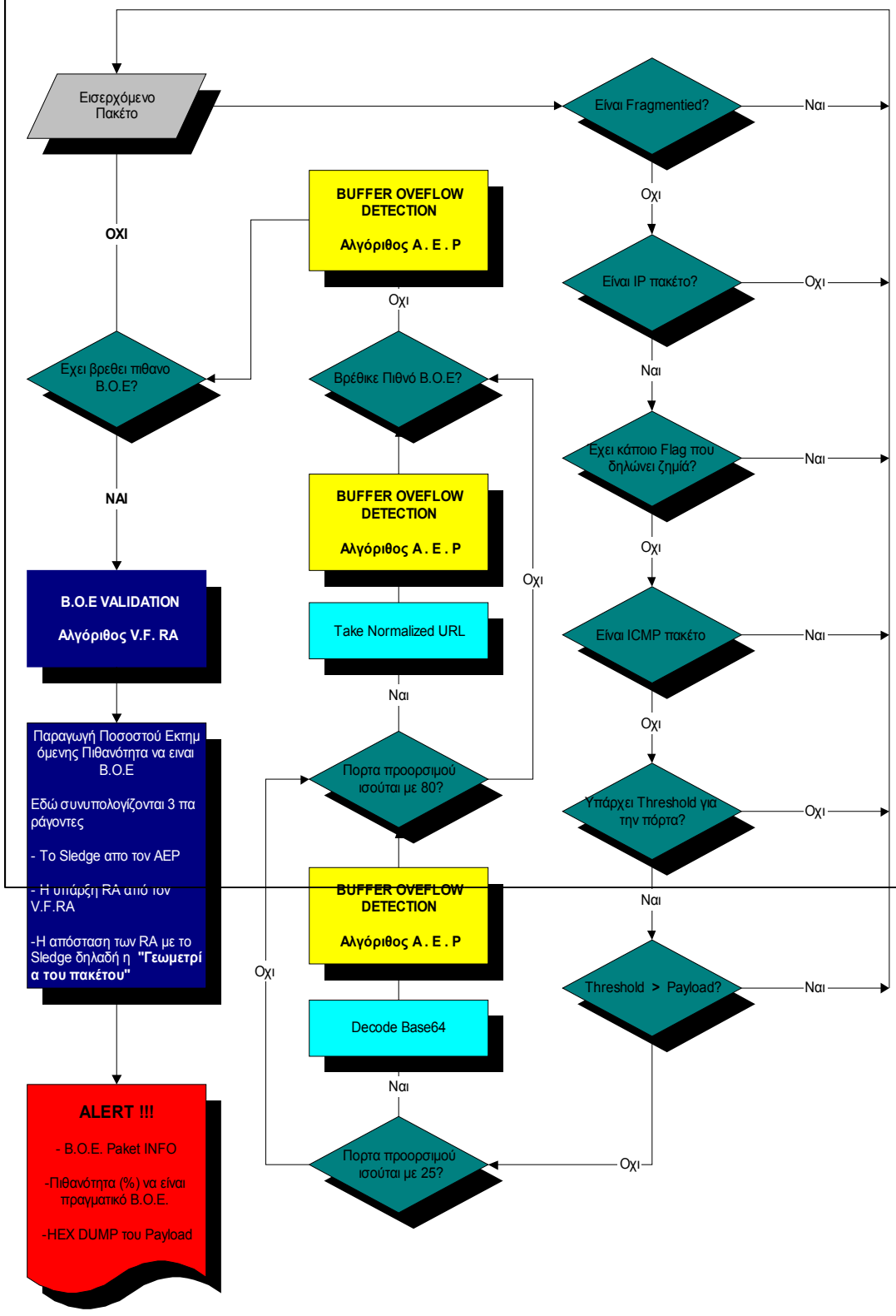
Οι συναρτήσεις που ακολουθούν κάνουν αναζήτηση για **ακολουθίες από RET** μέσα σε ένα πακέτο. Αυτές οι συναρτήσεις θα εκτελεστούν **μόνο αν ο AEP βρει πιθανό BOE**.

- long reverce_add(long);
- static int get_RA(unsigned char * bytes,int bytecode_size, int RA_threshold,long Oldposible_add,int pos,int number_of_RAs);
- static int test_for_RA(unsigned char * bytes,int bytecode_size, int RA_threshold);

Ομοίως με την περίπτωση του AEP αυτό ο αλγόριθμος που περιγράφεται αναλυτικά στο **κεφάλαιο 4**.

5.3 Η λειτουργία του Preprocessor

Στο σχήμα 5.1 παρακάτω παρουσιάζεται το διάγραμμα ροής του Preprocessor που στην πράξη αντιστοιχεί στην συνάρτηση **BuffOverDetectorFunction()**.


Εισερχόμενο Πακέτο
Είναι Fragmentied?
Ναι
Οχι
BUFFER OVEFLOW DETECTION
Αλγόριθος A . E . P
ΟΧΙ
Είναι IP πακέτο?
Οχι
Ναι
Έχει βρεθει πιθανο B.O.E?
Βρέθικε Πιθνό B.O.E?
Έχει κάποιο Flag που δηλώνει ζημιά?
Ναι
ΝΑΙ
BUFFER OVEFLOW DETECTION
Αλγόριθος A . E . P
Οχι
Οχι
B.O.E VALIDATION
Αλγόριθος V.F. RA
Take Normalized URL
Είναι ICMP πακέτο
Ναι
Ναι
Παραγωγή Ποσοστού Εκτημ όμενης Πιθανότητα να ειναι B.O.E
Εδώ συνυπολογίζονται 3 παράγοντες
- Το Sledge απο τον AEP
- Η υπάρξη RA από τον V.F.RA
-Η απόσταση των RA με το Sledge δηλαδή η "Γεωμετρία του πακέτου"
Πορτα προορσιμού ισούται με 80?
Υπάρχει Threshold για την πόρτα?
Οχι
Οχι
Ναι
BUFFER OVEFLOW DETECTION
Αλγόριθος A . E . P
Threshold > Payload?
Ναι
Οχι
Decode Base64
Ναι
ALERT !!!
- B.O.E. Paket INFO
-Πιθανότητα (%) να είναι πραγματικό B.O.E.
-HEX DUMP του Payload
Πορτα προορσιμού ισούται με 25?
Οχι

Σχήμα 5.1

5.4 Ο πηγαίος κώδικας του Preprocessor

ssp_BufferOverflowDetector.h

```
/* $Id$ */
/* Snort Preprocessor Plugin Header File*/

/* This file gets included in plugbase.h when it is integrated into the
rest
 * of the program.  Sometime in The Future, I'll whip up a bad ass Perl
script
 * to handle automatically loading all the required info into the
plugbase.*
 * files.
 */
#include "snort.h"
//#include <stdio.h> //Use it only if u will test it on shell

#include <stdbool.h> // Enables boolean type into a C program

#define NONE 0
#define REG  1
#define BV1  2
#define BV2  4
#define BV4  8
#define I1   16
#define I2   32
#define I4   64
#define REL8   128
#define REL16   256
#define REL32   512
#define REGREL8 1024
#define REGREL16 2048
#define REGREL32 4096

#define RESPONSE 0
#define DEFAULT_THRESHOLD 60

#ifndef __SPP_TEMPLATE_H__
#define __SPP_TEMPLATE_H__

struct trienode {
  int is_leaf;
  int number_of_operands;
  int * operands;
  struct trienode * triefollowers[256];
  char * line;
  char opcode_length;
  unsigned char jump;
};

typedef struct trienode * trieptr;

/* list of function prototypes for this preprocessor */
//List of function used for pluging-in to Short
void SetupBuffOverDetector();
void BuffOverDetectorInit(u_char *);
void ParseBuffOverDetectorArgs(char *);
void BuffOverDetectorFunction(Packet *);
void PreprocRestartFunction(int);
void PreprocCleanExitFunction(int);

/*list of functions for Buffer overflow detection*/
//static void error(char * string);
//Buliding "Trie" functions
static void checkEndOfString(char * string);
static char * getEndOfTokenMultiple(char * string, char * delimiters);
static char * getStartOfOpcode(char * string);
static char * getStartOfNextOpcode(char * string);
static void clear_node(trieptr ptr);
static trieptr new_trie_node(int nr_of_operands,int *ops, char * string,
int opcode_length, unsigned char jump);
trieptr buildTrie(char * filename);
static trieptr insertString(trieptr trie, int number_of_opcode_bytes,int
```

spp_BufferOverflowDetector.c

```
/* $Id$ */
/* Snort Preprocessor Plugin Source File BufferOverflowDetector */

/* spp_BufferOverflowDetector
 *
 * Purpose:
 * This Pre-processor is crated for detecting Hacking Attackes based on
 * Buffer Overflow vulnerabilities. The Basic Idea is to find how many
Instaction
 * this is in a sting. if the instacion chain in "long" enough then its
 * propable a Buffer Overflow exploit attempt.The Idea is based on the
 * "Abstract Execution Paper". The Idea say that the NOP sledges or NOP
 * equal OPs can cause long enough instracion chains (IC) so defnetly we
have
 * Buffer overflow exploit attempt.
 *
 *
 * Arguments:
 * The arguments are the ports we want to examin and the threshold.
 * Every port is bind to an Appilication. Every port change the
 * instracion chain length threshold for which the specifig appication
 * we don't have a Buffer Over flow trying.
 *     *Format: port;thershold
 *     *ports available: 80, 8080, 21 e.t.c.
 *
 *
 * Effect: It has no effect on the Packet(s)
 *
 *
 * Comments:
 * The BufferOverflow Detector Check ONLy the PentiumI instacion set.
 * Always tune the Threshold based on your NetWork enviroment.
 * Enfasize on Windows based sevicies
 *
 *
 * NOTE:
 * The basic C implemtation for "abstract executio payload" inspaction
 * method originaly was found it at the "Apache code detector module -
detects buffer overflow exploits"
 * witch prevents malicious requests on Apache server.
 * Author of the apache module is : Thomas Toth ,
ttoth@infosys.tuwien.ac.at
 *
 *
 * AUTHOR: Dimitris Pritsos, dpritsos@lab.epmhs.gr
 *
 * TODO:
 * Add other Proccesors instracion sets.
 *
 *
 */

/*Header files Buffer Overflow Detector needs*/
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#ifdef HAVE_STRINGS_H
#include <strings.h>
#endif

#include <sys/types.h>

#include "decode.h"
#include "plugbase.h"
#include "parser.h"
#include "log.h"
#include "debug.h"
#include "util.h"
```