Κεφάλαιο 3
Μέθοδοι αντιμετώπισης των Buffer Overflow Exploits

3.1 Γενικά

Στο Α Μέρος της πτυχιακής παρουσιάσθηκε αναλυτικά το πώς μπορεί ένα **κακόβουλος χρήστης(Blackhat)** να **εκμεταλλευτεί(Exploit)** την **αδυναμία(Vulnerability) Buffer Overflow** μίας εφαρμογή ώστε να αποκτήσει **μη εξουσιοδοτημένη(Unauthorized)** πρόσβαση σε ένα υπολογιστικό σύστημα. Ακόμα ειπώθηκε ότι το πρόβλημα ασφάλειας που προκαλεί αυτή **η προγραμματιστική αδυναμία** είναι γνωστή τουλάχιστον 15 χρόνια. Αυτά τα χρόνια έχουν γίνει μεγάλες και αξιόλογες προσπάθειες ώστε να το πρόβλημα αντιμετωπιστεί με διάφορες μεθόδους. Οι μέθοδοι αυτοί μπορούν να καταταχτούν, αναλόγως με τον τρόπο που αντιμετωπίζουν το πρόβλημα, στις μεθόδους **Πρόληψης** και στις μεθόδους **Καταστολής**.

Οι μέθοδοι που ο σκοπός τους είναι η **πρόληψη** προσπαθούν να εμποδίσουν να συμβεί ένα **Buffer Overflow** σε μία εφαρμογή. Αυτό έχει σαν συνέπεια ότι η εφαρμογή είναι απαλλαγμένη τουλάχιστον από τον κίνδυνο **επιθέσεων** που βασίζονται σε Buffer Overflow αδυναμίες.

Οι μέθοδοι της **καταστολής** δεν ασχολούνται με την ίδια την εφαρμογή ή την διαδικασία ανάπτυξης της. Οι μέθοδοι αυτοί χρησιμοποιούνται ώστε να προστατέψουν το **υπολογιστικό** ή **ολόκληρο το πληροφοριακό σύστημα** αφού συμβεί η επίθεση. Αυτή η κατηγορία των μεθόδων όπως εύκολα μπορεί να διαπιστώσει κανείς διαβάζοντας την σχετική παράγραφο παρακάτω, σαν κύριο σκοπό έχουν να προστατεύουν τα συστήματα μέχρι αυτά να απαλλαγούν από τις αδυναμίες Buffer Overflow.

Σε αυτό το κεφάλαιο θα παρουσιαστούν οι μέθοδοι αντιμετώπισης του προβλήματος **των Buffer Overflow Exploits** παρουσιάζοντας πρώτα την κατηγορία των μεθόδων πρόληψης και μετά αυτό της καταστολής. Αυτό γίνεται για να μπορεί ο αναγνώστης να έχει μία οριζόντια γνώση για το πώς έχει αντιμετωπιστεί αυτό το πρόβλημα πριν καταλάβει σε βάθος της μέθοδο που υιοθετείται σε αυτή την πτυχιακή.

3.2 Μέθοδοι πρόληψης του Buffer Overflow

Στις μεθόδους που έχουν προταθεί σε αυτή την κατηγορία το βασικό χαρακτηριστικό στο οποίο βασίζονται είναι ο **έλεγχος ορίων(Boundary Checking)**. Αυτό φυσικά είναι αναμενόμενο αφού από την ακούσια ή εκούσια παράληψη αυτού του ελέγχου ξεκινά η ρίζα του κακού. Οι πιο διαδεδομένοι μέθοδοι **πρόληψης** του προβλήματος των **Buffer Overflow Exploit** είναι :

- **Συγγραφή secure code**
- **Dynamic run-time checks - Libsafe**
- **Automated Detection of Buffer Overflow via Static Analysis**
- **Brute-Force Analysis**

3.2.1 Συγγραφή secure code

Η συγγραφή **Ασφαλή Κώδικα(Secure Code)** είναι ο τρόπος αντιμετώπισης του προβλήματος που προτείνεται από όλα τα όλες τις **εργασίες** και τις **δημοσιεύσεις** που έχουν γίνει μέχρι σήμερα(2003). Αυτή η **λύση είναι περισσότερο μία πρόταση** για την εφαρμογή μιας **πολιτικής** πάνω στην ανάπτυξη κώδικα χωρίς τέτοιου είδους λάθη.

Η εφαρμογής μίας τέτοιας **πολιτικής** πρέπει να γίνεται σε δύο φάσεις. Η πρώτη φάση είναι ότι μία εταιρία παραγωγής λογισμικού να θεωρεί ότι μέρος της εκπαίδευσης των προγραμματιστών της θα είναι η γνώση της συγγραφής κώδικα με όσο το δυνατόν πιο τυποποιημένο τρόπο. Αυτό θα οδηγήσει τους προγραμματιστές να αποφεύγουν να χρησιμοποιούν μεθόδους ή βιβλιοθήκες που είναι γνωστό ότι υποφέρουν από Buffer Overflow αλλά και όταν τις χρησιμοποιούν να δίνουν την απαραίτητη προσοχή. Το δεύτερο κομμάτι αφορά τον σχεδιασμό κάθε έργου και την **διαδικασία ελέγχων ποιότητας** που πρέπει να γίνονται στο λογισμικό. Όσο αφορά τον σχεδιασμό πρέπει από πριν να έχει αποφασιστεί τι εργαλεία θα χρησιμοποιηθούν και από ποίες γνωστές **αδυναμίες Buffer Overflow υποφέρουν ώστε να απαλειφθούν**. Τέλος όσο αφορά τους **έλεγχους ποιότητας** πρέπει

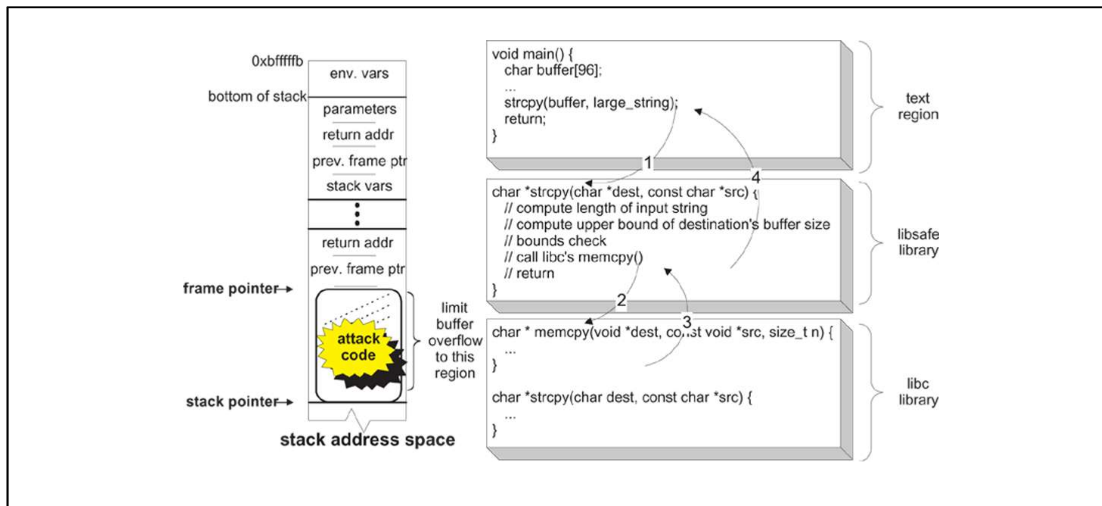
κυρίως να ελέγχουν πόσο συνεπείς μπόρεσαν να είναι οι προγραμματιστές στην τυποποιημένη διαδικασία που τους ζητήθηκε και να στην συνέχεια να γίνεται εκτεταμένη ανάλυση σε κάθε «παρατυπία». Για παράδειγμα αν είναι συμφωνημένο κατά την τυποποίηση της ότι δεν θα χρησιμοποιείται η συνάρτηση **scanf()** αλλά αυτή για κάποιο λόγο είναι υποχρεωτικό να χρησιμοποιηθεί τότε θα πρέπει να γίνει εκτεταμένος έλεγχος ώστε η αυτή να μην προκαλέσει κάποια από τις γνωστές αδυναμίες.

Το πρόβλημα της παραπάνω λύσης είναι η αδυναμία του ανθρώπου να αντιστέκεται στα λάθη. Έτσι είναι στατιστικά αποδεδειγμένο ότι ακόμα και αν μία διαδικασία παραγωγής λογισμικού έχει προβλέψει όλους τους πιθανούς τρόπους που μπορεί κάποιος να επιτεθεί στο **προϊόν λογισμικό** της είναι πολύ πιθανό να υπάρχει και ένας τουλάχιστόν ακόμα τρόπος που μπορεί να επιτεθεί κάποιος σε αυτό.

Για να λυθεί αυτό το πρόβλημα έχουν αναπτυχθεί προγραμματιστικά εργαλεία που ή ενσωματώνουν στον κώδικα τους **απαραίτητους ελέγχους ορίων** ή ελέγχουν και προειδοποιούν τον προγραμματιστή ότι ο κώδικάς του έχει κάποιες αδυναμίες. Οι πιο διαδεδομένες λύσεις παρουσιάζονται στις επόμενες παραγράφους.

3.2.2 Dynamic run-time checks – Libsafe

Η **Libsafe** αναπτύχθηκε στα **Bell Labs** και ο βασικός της στόχος είναι να αντικαταστήσει όλες τις συναρτήσεις της **Libc** (κεφάλαιο 2 - stdio.h) που έχουν **Buffer Overflow** αδυναμίες. Ο τρόπος που λειτουργεί η **Libsafe** δεν αντικαθιστά την **Libc** αλλά προσθέτει στις **αδύναμες συναρτήσεις Boundary Checking** όπως φαίνεται στο *Σχήμα 3.1.*



Σχήμα 3.1

Μετρήσεις στο σχετικό Paper που γράφτηκε για την LibSafe δείχνουν ότι στην πράξη μία εφαρμογή που χρησιμοποιεί την Libsafe έχει την **ίδια περίπου απόδοση(Performance)** με την έκδοση της εφαρμογής που δεν την χρησιμοποιεί.

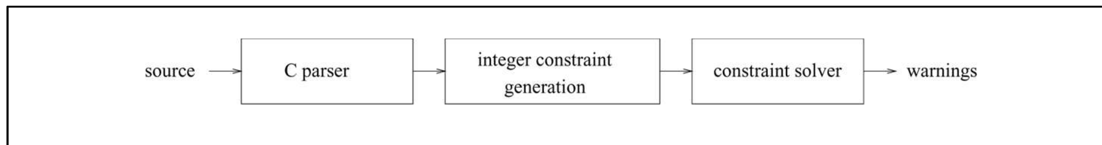
Το μεγαλύτερο πλεονέκτημα αυτής της λύσης είναι ότι οι περισσότερες νέες εφαρμογές **δεν χρειάζεται να ξαναγίνουν compile** λόγο του Dynamic Linking. Το σημαντικότερο της μειονέκτημα είναι ότι **προστατεύει μία εφαρμογή μόνο από την λανθασμένη χρήση των συναρτήσεων της βιβλιοθήκης Libc** μη μπορώντας να απαλείψει όλες τις υπόλοιπες αδυναμίες Buffer Overflow.

Ενώ η λύση τις **Libsafe** είναι αρκετά ικανοποιητική, η ανάγκη να βρεθεί άλλη λύση που θα αντιμετωπίζει το πρόβλημα καθολικά οδήγησε σε **λύσεις εργαλείων** που βοηθούν το προγραμματιστή κατά την διάρκεια του **Debugging** να ανακαλύπτει που έχει αφήσει Buffer Overflow αδυναμίες. Μία από αυτές τις λύσεις παρουσιάζεται στην επόμενη παράγραφο.

3.2.3 Automated Detection of Buffer Overflow via Static Analysis

Το πανεπιστήμιο **Berkeley** δημοσίευσε το paper **"A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities"** που περιγράφει μία μέθοδο ανάλυσης του κώδικα ώστε να απαλείφονται τα Buffer overflow. Η μέθοδος αυτή στην πράξη ανάγει το πρόβλημα σε πρόβλημα **περιορισμών ακέραιων αριθμών(Integer Constraint)**. Η αναγωγή αυτή έχει δύο πλεονεκτήματα. Το πρώτο είναι ότι το πρόβλημα του **Buffer Overflow** μπορεί να λυθεί με οποιοδήποτε αλγόριθμο που λύνει το μαθηματικό πρόβλημα **Integer Constraint**. Το δεύτερο είναι ότι η λύση αυτή μπορεί να εφαρμοστεί σε πολύ μεγάλες εφαρμογές δίνοντας λύση σε **μικρό πολυωνυμικό χρόνο** επειδή οι πράξεις σε αριθμούς γίνονται πολύ γρήγορα.

Η λύση αυτή εφαρμόζεται κατά το Compilation παράγοντας ένα σύνολο από Warnings που ειδοποιούν τον προγραμματιστή για της αδυναμίες Buffer Overflow που εντοπίσθηκαν. Στο παρακάτω σχήμα παρουσιάζεται ο τρόπος που συνεργάζεται η λύση αυτή με ένα Compiler.



Σχήμα 3.2

Η βασική δυσκολία που έχει να αντιμετωπίσει αυτή η λύση είναι **η μοντελοποίηση όλων των** δυνατών **προτύπων** πηγαίου κώδικα που μπορούν να προκαλέσουν Buffer Overflow αδυναμίες. Σε αυτή την δυσκολία οφείλεται και η αδυναμία αυτής της λύσης να βρει όλες τις πιθανές περιπτώσεις **Buffer Overflow Vulnerabilities**. Στην πράξη με βάση τις μέτρησεις που έγιναν στο Paper, οι περιπτώσεις που δεν μπορεί να συλλάβει είναι όταν το Buffer δεν δηλώνεται με στατικό μέγεθος αλλά αρχικά δηλώνεται μόνο ο Pointer. Οι αδυναμίες που μπορεί να προκύψουν από αυτή την περίπτωση είναι σαν αυτές της περίπτωσης της κακής χρήσης της Malloc στο **Παράρτημα A.1.**

Η περαιτέρω επεξήγηση αυτής της λύσης ξεφεύγει από τα όρια αυτή της πτυχιακής για αυτό το λόγο παρακάτω παρουσιάζεται ένας πίνακας που δείχνει ένα δείγμα της διαδικασίας μοντελοποίησης. Μετά την **μοντελοποίηση** και την παραγωγή μαθηματικών περιορισμών από τα μοντέλα των συναρτήσεων μπορεί να χρησιμοποιηθεί ένας οποιοσδήποτε αλγόριθμος **constraint solver** όπως φαίνεται στο *σχήμα 3.2*.

C code	Interpretation
`char s[n];`	$n \subseteq \text{alloc}(s)$
`strlen(s)`	$\text{len}(s) - 1$
`strcpy(dst,src);`	$\text{len}(src) \subseteq \text{len}(dst)$
`strncpy(dst,src,n);`	$\min(\text{len}(src), n) \subseteq \text{len}(dst)$
`s = "foo";`	$4 \subseteq \text{len}(s), \quad 4 \subseteq \text{alloc}(s)$
`p = malloc(n);`	$n \subseteq \text{alloc}(p)$
`p = strdup(s);`	$\text{len}(s) \subseteq \text{len}(p), \quad \text{alloc}(s) \subseteq \text{alloc}(p)$
`strcat(s,suffix);`	$\text{len}(s) + \text{len}(suffix) - 1 \subseteq \text{len}(s)$
`strncat(s,suffix,n);`	$\text{len}(s) + \min(\text{len}(suffix) - 1, n) \subseteq \text{len}(s)$
`p = getenv(...);`	$[1, \infty] \subseteq \text{len}(p), \quad [1, \infty] \subseteq \text{alloc}(p)$
`gets(s);`	$[1, \infty] \subseteq \text{len}(s)$
`fgets(s,n,...);`	$[1, n] \subseteq \text{len}(s)$
`sprintf(dst,"%s",src);`	$\text{len}(src) \subseteq \text{len}(dst)$
`sprintf(dst,"%d",n);`	$[1, 20] \subseteq \text{len}(dst)$
`snprintf(dst,n,"%s",src);`	$\min(\text{len}(src), n) \subseteq \text{len}(dst)$
`p[n] = '\0';`	$\min(\text{len}(p), n + 1) \subseteq \text{len}(p)$
`p = strchr(s,c);`	$p = s+n; \quad [0, \text{len}(s)] \subseteq n$
`h = gethostbyname(...);`	$[1, \infty] \subseteq \text{len}(h\text{->}h_name),$ $[-\infty, \infty] \subseteq h\text{->}h_length$

3.2.4 Brute-force Analysis

Μία μέθοδο που χρησιμοποιείται κατά κόρον από τους **Blackhat Hackers** για να βρίσκουν αδυναμίες **Buffer Overflow** σε ένα σύστημα είναι η **Brute-force Analysis**. Αυτή τη μέθοδο τείνουν τα τελευταία χρόνια να χρησιμοποιούν και **οι Διαχειριστές(Administrators) των**

δικτύων για να ελέγχουν αν τα υπολογιστικά συστήματα που υπάρχουν στο δίκτυο τους έχουν αδυναμίες τέτοιου τύπου.

Η μέθοδος **Brute-force Analysis** δεν είναι μία μέθοδος που χρησιμοποιείται μόνο για Buffer Overflow, αντίθετα μάλιστα την μέθοδο αυτή την χρησιμοποιούν για όλες τις πιθανές αδυναμίες που μπορεί έχει το σύστημα. Στην πράξη η μέθοδος αυτή χρησιμοποιείται από ένα σύνολο **εργαλείων λογισμικού** που **πραγματοποιούν όλες τις γνωστές επιθέσεις που υπάρχουν σε ένα σύστημα**. Αυτά τα εργαλεία όμως αντί να «καταλάβουν» τελικά το σύστημα απλά αναφέρουν στον διαχειριστή του δικτύου ποία από τα συστήματα του δικτύου του έχουν κάποιο πρόβλημα και ποίο είναι αυτό.

Η μέθοδος αυτή αν και εφαρμόζεται σε μία εφαρμογή **αφού αυτή έχει μπει στην φάση της συντήρησης του κύκλου ζωής της** θεωρείται προληπτική μέθοδος γιατί εφαρμόζεται πριν το σύστημα δεχτεί πραγματικά την επίθεση. Η μέθοδος αυτή όμως έχει το μεγάλο μειονέκτημα ότι λόγο των δοκιμών για τυχών αδυναμίες μπορεί να προκαλέσει μεγάλη ζημιά στο υπολογιστικό σύστημα. Τέλος, το πλεονέκτημα της είναι ότι μπορεί να βρίσκει αδυναμίες σε συστήματα που είναι έτοιμα να μπουν στην παραγωγική διαδικασία, βοηθώντας τον διαχειριστή του δικτύου να απαλείψει πολλές από της αδυναμίες του δικτύου του πριν αυτές του δημιουργήσουν πραγματικό πρόβλημα.

3.3 Μέθοδοι αντιμετώπισης του Buffer Overflow μέσω της προστασίας κρίσιμων πληροφοριών

Πριν παρουσιαστούν οι μέθοδοι καταστολής του προβλήματος του Buffer Overflow σε αυτή την παράγραφο θα παρουσιαστούν τρεις από τις μεθόδους που για να εμποδίσουν τα Buffer Overflow Exploit διαφυλάσσουν τις κρίσιμες για την ροή του προγράμματος πληροφορίες. Οι μέθοδοι αυτοί δεν κατατάσσονται σε καταστολή ή πρόληψη γιατί ο τρόπος που λειτουργούν περιλαμβάνει χαρακτηριστικά και από τις δύο κατηγορίες.

Στην πράξη η ιδέα των μεθόδων αυτών είναι να ελέγχουν αν κάποια κρίσιμη πληροφορία όπως για παράδειγμα η **RA(Return Address)** μίας συνάρτησης αλλάχθηκε από εξωτερικό παράγοντα. Αν διαπιστωθεί μία τέτοια αλλαγή τότε σταματά η εκτέλεση του προγράμματος. Για να μπορεί όμως να γίνεται αυτός ο έλεγχος πρέπει το πρόγραμμα να έχει γίνει Compiled με ένα Compiler που διαθέτει αυτό το μηχανισμό. Προφανώς οι μέθοδοι πρέπει να εφαρμοστούν σε μία εφαρμογή **προληπτικά**, κατά το Compilation, ενώ ο μηχανισμός που ενσωματώνεται στο πρόγραμμα για να σταματά τα BOE λειτουργεί **κατασταλτικά**. Η λειτουργία των μεθόδων αυτών θεωρείται κατασταλτική γιατί σταματούν την ροή του προγράμματος αφού συμβεί το **Buffer Overflow** χωρίς να προσπαθούν να το εμποδίσουν πριν αυτό συμβεί.

Η κάθε μία από τις παρακάτω μεθόδους χρησιμοποιεί κάποιο μηχανισμό για να προστατεύσει το σύστημα από Buffer Overflow. Οι μέθοδοι αυτοί μπορούν να εφαρμοστούν με διάφορους τρόπους. Για παράδειγμα σε άλλες περιπτώσεις ο μηχανισμός γίνεται μέρος του Compiler ενώ σε άλλες περιπτώσεις απλά ο Compiler χρησιμοποιεί ένα C ή Assembly πρόγραμμα και το ενσωματώνει στον κώδικα μίας εφαρμογής. Ο τρόπος που εφαρμόζεται ο μηχανισμός δεν θα αναλυθεί στα πλαίσια αυτή της πτυχιακής γιατί δεν έχει τόσο σημασία όσο το τι κάνει ο μηχανισμός προστασία για να σταματήσει τα Buffer Overflow Exploits.

3.3.1 StackGuard

Η μέθοδος **StackGuard** εστιάζει την λύση του προβλήματος στην προστασία της **RA** μίας function. Για να το κάνει αυτό χρησιμοποιεί μια **32bit λέξη** σαν μηχανισμό προστασίας της RA. Η βασική ιδέα είναι ότι αν στην θέση που βρίσκεται αυτή η λέξη μέσα στην στοίβα βρεθεί κάποια άλλη λέξη τότε θεωρείται ότι έγινε υπερχείλιση και σταματά η εκτέλεση του προγράμματος.
Για να μπορεί να ελεγχθεί κάθε φορά αν η **λέξη** αλλάχθηκε ενσωματώνεται, από τον Compiler, στην κλήση κάθε συνάρτησης ένα ζευγάρι **προλόγου** και ένα **επιλόγου** για την εφαρμογή του StackGuard μηχανισμού. Ο **πρόλογος του StackGaurd** δεν αντικαθιστά τον **πρόλογο** μίας συνάρτησης αλλά προστίθεται πριν από αυτόν ενώ ο επίλογος του **StackGaurd προστίθεται** αμέσως μετά από τον επίλογο της. Στο παρακάτω παράδειγμα φαίνεται πώς διαμορφώνεται τελικά ο κώδικας μίας συνάρτησης μετά το Compilation.

Παράδειγμα 3.1

```
function_prologue:
      pushl $0x000aff0d // push canary into the stack
      pushl %ebp // save frame pointer
      mov %esp,%ebp // saves a copy of current %esp
      subl $108, %esp // space for local variables

      .
      .
(function body)
      .
      .
      .
function_epilogue:
      leave // standard epilogue
      cmpl $0x000aff0d,(%esp) // check canary
      jne canary_changed
      addl $4,%esp // remove canary from stack
      ret

canary_changed:
      ... // abort the program with error
      call __canary_death_handler
      jmp . // just in case I guess
```

Στο παράδειγμα φαίνεται ότι η **32bit λέξη που λέγεται Canary** είναι μια **σταθερή τιμή** που τοποθετείται ακριβώς πάνω από την **Return Address**. Με αυτό τον τρόπο αν ένα Buffer Overflow Exploit υπερχειλίσει το Buffer τότε πριν αντικαταστήσει την **RA** θα έχει ήδη αλλάξει την τιμή του **Canary Word** έτσι ο StackGaurd μηχανισμός θα καταλάβει την υπερχείλιση και θα σταματήσει την εκτέλεση του προγράμματος. Για να γίνει αυτό κατά την **επιστροφή της συνάρτησης** πριν ακριβώς την αποκατάσταση του EIP με την RA θα εκτελεστεί ο επίλογος του **StackGaurd**. Ο επίλογος αυτός θα ελέγξει αν το Canary είναι **άθικτο** και αν όχι θα σταματήσει την εκτέλεση του προγράμματος πριν εκτελεστεί το Buffer Overflow Exploit.

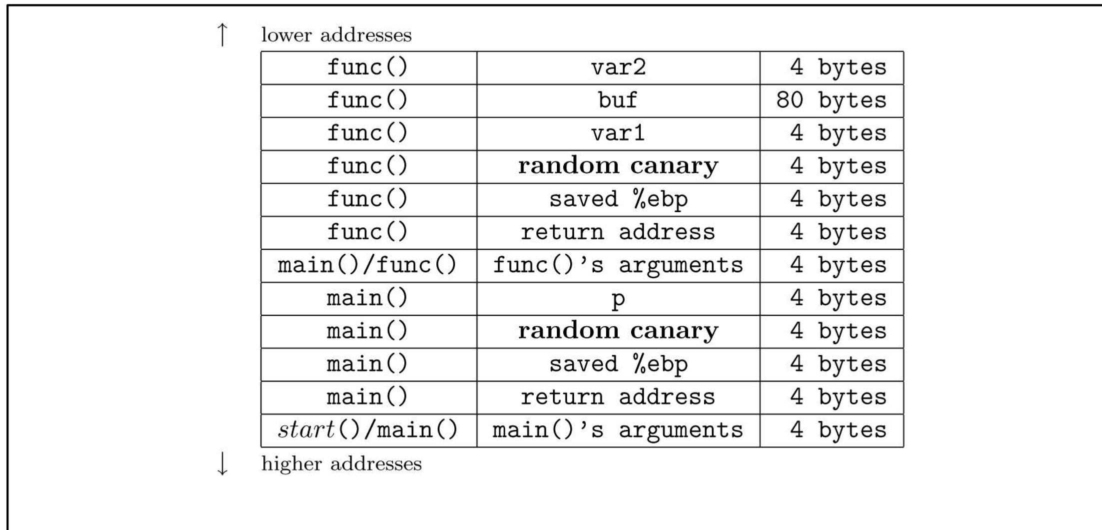
Η λύση αυτή αρχικά ήταν τελείως λανθασμένη γιατί το μόνο που έπρεπε να κάνει ένας **Blackhat** για να παρακάμψει αυτή τη δικλείδα ασφαλείας ήταν πριν από κάθε RET μέσα στο BOE String να προσθέσει το Canary αφού η τιμή αυτή είναι στατική. Για να λυθεί αυτό το πρόβλημα στις επόμενες εκδόσεις του το StackGaurd αντί να χρησιμοποιεί ένα στατικό Canary χρησιμοποιεί ένα **Random Canary** και για να αυξηθεί η εντροπία του αυτό ο Canary κάνει την πράξη XOR με την RA της εκάστοτε Function. Την λύση αυτή την υιοθετεί ο **Visual C/C++ compiler του .NET Studio της Microsoft**.

3.3.2 Visual C/C++ compiler του .NET Studio της Microsoft

Η Microsoft επέλεξε να χρησιμοποιεί ένα παραπλήσιο μηχανισμό με αυτό του **StackGaurd με Random Canary** στον Compiler του **.NET** ώστε οι επόμενες εφαρμογές που θα παραχθούν για την πλατφόρμα αυτή να είναι «ασφαλέστερες». Η ιδέα ενός Random Canary είναι πολύ σπουδαία γιατί ο Blackhat θα πρέπει να βρει έναν μηχανισμό που θα προβλέπει τα **Canaries** που μπορεί ο Compiler να παράγει. Ακόμα και αν ξέρει τον μηχανισμό μόνο μετά από μία διαδικασία **Brute-force** θα μπορέσει τελικά να πετύχει τον σκοπό του.

Στην λύση αυτή όμως υπάρχει ένα μεγάλο πρόβλημα. Όπως ειπώθηκε στο **κεφάλαιο 2** η RA δεν είναι η μοναδική πληροφορία που αφορά την ροή του προγράμματος και σώζεται στη στοίβα. Αμέσως πριν από την RA υπάρχει ο **Frame Pointer** της στοίβας που και αυτό μπορεί να χρησιμοποιηθεί ώστε να αλλάξει την ροή του προγράμματος και να την στρέψει προς τον Shellcode. Φυσικά ο υπολογισμός του FP είναι πιο δύσκολος γιατί αυτός με την σειρά του πρέπει να οδηγήσει σε μια RA που θα προκαλέσει την αλλαγή της ροή. Για να μπορέσει

λοιπόν η **Microsoft** να καλύψει και αυτές τις περιπτώσεις φρόντισε να παραλλάξει λίγο το StackGaurd και να τοποθετεί το **Random Canary** πάνω από τον FP αντί πάνω από την RA όπως φαίνεται και στο *σχήμα 3.3.*



↑ lower addresses

func()	var2	4 bytes
func()	buf	80 bytes
func()	var1	4 bytes
func()	**random canary**	4 bytes
func()	saved %ebp	4 bytes
func()	return address	4 bytes
main()/func()	func()'s arguments	4 bytes
main()	p	4 bytes
main()	**random canary**	4 bytes
main()	saved %ebp	4 bytes
main()	return address	4 bytes
start()/main()	main()'s arguments	4 bytes

↓ higher addresses

Σχήμα 3.3

Η μέθοδος αυτή είναι αρκετά ασφαλέστερη από την λύση του **StackGaurd,** αλλά το πρόβλημα ανάγεται στο να μάθει να υπολογίζει τα **Random Canaries** ώστε να εφαρμόζει όλα τα **Buffer Overflow** που είναι σχεδιασμένο να τρέχουν όταν δεν υπάρχει ο αυτός ο προστατευτικός μηχανισμός.

3.3.3 StackShield

Η ιδέα του **StackShield** προσπαθεί να λύσει τις αδυναμίες που έχουν οι παραπάνω λύσεις δημιουργώντας ένα απλούστερης λογικής μηχανισμό. Η ιδέα είναι ο πρόλογος **που προστίθεται σε μία συνάρτηση για να την προστατέψει**, να αντιγράφει την RA στον σωρό αντί να βάζει **Canary** μέσα στην στοίβα. Έτσι, ο επίλογος το μόνο που θα πρέπει να κάνει είναι να ελέγχει αν η **διεύθυνση επιστροφής** είναι ή ίδια με αυτή που αντιγράφτηκε αρχικά στον σωρό ή άλλαξε. Είναι προφανές ότι ο Blackhat με αυτό τον τρόπο δεν μπορεί να κάνει τίποτα για να αντιμετωπίσει τον μηχανισμό παρά μόνο να επέμβεί στο σημείο του σωρού που βρίσκεται το αντίγραφο της RA και να το αντικαταστήσει. Όπως φαίνεται λοιπόν ακόμα και σε αυτή την λύση υπάρχει τρόπος να «εξαπατηθεί» ο μηχανισμός προστασίας και τελικά το BOE να πετύχει τον στόχο του.

Αν και οι μηχανισμοί αυτοί δεν λύνουν καθολικά το πρόβλημα τουλάχιστον αυξάνουν τον βαθμό ασφάλειας μία εφαρμογής. Όμως το μεγαλύτερο πρόβλημα που μπορεί να προκαλέσει η χρήση αυτών των μηχανισμών είναι η ψευδαίσθηση ασφάλειας στους προγραμματιστές. Η βασική αιτία που δημιουργείται αυτή η ψευδαίσθηση είναι γιατί οι περισσότεροι αγνοούν ότι τα Buffer Overflow δεν γίνονται μόνο στην **στοίβα** αλλά και στον **σωρό** ενώ παράλληλα οι παραπάνω μηχανισμοί εστιάζουν μόνο σε αυτή την πλευρά του προβλήματος. Έτσι τα **Heap Based Buffer Oveflows** αδυνατούν να τα εντοπίζουν αφήνοντας **exploits** όπως το **openSSL remote exploit** που παρουσιάζεται αναλυτικά στο Παράρτημα A.1 να εκμεταλλευτούν ανενόχλητα αυτές τις αδυναμίες.

3.4 Μέθοδοι καταστολής του Buffer Overflow

Εκτός από τις λύσεις των δύο προηγούμενων παραγράφων όπως ειπώθηκε από την αρχή του κεφαλαίου υπάρχουν και οι λύσεις της **καταστολής**. Οι λύσεις αυτές σε αντίθεση με το **StackGuard** ή τον Compiler της Microsoft εφαρμόζονται συμπληρωματικά χωρίς να χρειάζεται να ενσωματωθούν κατά το Compilation. Οι λύσεις αυτές δεν έχουν πάντα στόχο να αποτρέψουν την εκτέλεση των **Buffer Overflow exploit** αλλά τουλάχιστον να ειδοποιήσουν το χρήστη ή διαχειριστή ενός συστήματος ότι το σύστημά του δέχτηκε επίθεση ώστε αυτό να αντιδράσει κατάλληλα. Στην παράγραφο αυτή λοιπόν πρώτα θα παρουσιαστούν οι γνωστότεροι μέθοδοι που κατασταλτικά αποτρέπουν την εκτέλεση των **Buffer Overflow**

Exploits και στη συνέχεια αυτοί που **περιμένουν την ανθρώπινη παρέμβαση** για να αντιδράσουν σε ένα Buffer Overflow Exploit.

3.4.1 Μη εκτελέσιμη στοίβα

Η περιοχή του κώδικα ενός προγράμματος έχει **δικαιώματα μόνο εκτέλεσης** ακόμα και για την ίδια την εφαρμογή. Έτσι δεν μπορεί να γίνει καμία αλλαγή στον κώδικα ενός προγράμματος. Αντίθετα στην **στοίβα** τα δικαιώματα τυπικά είναι μόνο **Read/Write** που σημαίνει ότι τα δεδομένα που βρίσκονται μέσα σε **σελίδες(Pages)** δεν μπορούν να εκτελεστούν ακόμα και αν η ροή ενός προγράμματος στέφει προς αυτά.

Στην πράξη **η δυνατότητα η στοίβα να είναι μη εκτελέσιμη** δεν χρησιμοποιείται στα λειτουργικά συστήματα για λόγους **ευελιξίας** και **ταχύτητας**. Έτσι προγράμματα όπως ο GCC Compiler αλλά και πολλά λειτουργικά χαρακτηριστικά των Windows χρησιμοποιούν αυτή την δυνατότητα. Με αυτό τον τρόπο για παράδειγμα ο GCC παράγει τον κώδικα μίας συνάρτησης που θέλει να εκτελέσει μέσα στην στοίβα και στην συνέχεια στρέφει την ροή του προγράμματος προς την συνάρτηση αυτή.

Μία ιδέα που χρησιμοποιείται σε μερικές εκδόσεις του πυρήνα του Linux είναι να χρησιμοποιηθεί η δυνατότητα του Paging που προσφέρεται ώστε η στοίβα να μην είναι εκτελέσιμη. Η λύση αυτή είναι **προφανώς κατασταλτική** γιατί δεν εμποδίζει το Buffer Overflow να συμβεί αλλά **το Buffer Overflow Exploit να εκτελεστεί**.

Το πρόβλημα αυτής της λύσης είναι ότι αυτή μπορεί να προκαλέσει δυσάρεστες παρενέργειες στην λειτουργία ενός συστήματος που **οι εφαρμογές του χρησιμοποιούν την εκτελεσιμότητα της στοίβας**. Ακόμα όμως και αν λυθούν τα προβλήματα αυτά υπάρχει ακόμα η δυνατότητα να εκτελεστούν άλλες επιθέσεις στον σωρό όπως στο παράδειγμα **openSSL remote exploit** του *κεφαλαίου 3.*

Τελειώνοντας την περιγραφή αυτής της λύσης απλά θα αναφερθεί ότι υπάρχουν πολλές περίπτωσης που **μη εκτελεσιμότητα της στοίβας** παρακάμπτεται με την χρήση του σωρού σε συνδυασμό διαφόρων άλλων χαρακτηριστικών της λειτουργία των υπολογιστικών συστημάτων όπως είναι το GOT table.

3.4.2 Anti-Virus

Αρκετά χρόνια πριν εμφανιστεί το **Worm του Morris** στην κοινότητα των υπολογιστών εμφανίστηκαν οι **Ioί(virus)** των υπολογιστών. Οι γνωστοί σε όλους **Virus** δεν εκμεταλλεύονταν μόνο αδυναμίες Buffer Overflow αλλά και πολλές άλλες. Για να αντιμετωπιστεί αυτό το πρόβλημα τότε αναπτύχθηκαν εφαρμογές που ο σκοπός τους είναι να εντοπίζουν και να καταστρέφουν **προγράμματα ιούς**. Οι εφαρμογές αυτές λέγονται **Antivirus** όπως το **Norton-Antivirus** ή το **Mcaffe** κ.τ.λ.

Κάποτε η χρήση των Antivirus ήταν θέμα επιλογής ανάλογα με την κρισιμότητα των πληροφοριών που έπρεπε να διαφυλαχτούν από πιθανές ζημίες που μπορεί να προκαλέσει ένα Virus. Σήμερα όμως είναι σχεδόν απαραίτητο σε κάθε υπολογιστικό σύστημα που **θα χρησιμοποιηθεί σαν περιβάλλον μακροχρόνιας εργασίας.** Η ανάγκη αυτή προκύπτει από την μεγάλη πολυπλοκότητα που έχουν τα σημερινά πληροφοριακά συστήματα καθώς και την **ποικιλία** των ιών.

Τα **AntiVirus** λοιπόν όπως και τα **Virus** εξελίχθηκαν και σήμερα δίνουν ένα καλό επίπεδο προστασίας σε ένα υπολογιστικό σύστημα. Το σημαντικότερο από όλα όμως είναι ότι ένα **AntiVirus** μπορεί να προστατέψει κατασταλτικά ένα σύστημα από επιθέσεις Buffer Overflow χωρίς να **αλλάξει την δομή και λειτουργία του συστήματος** όπως οι λύσεις που παρουσιάστηκαν μέχρι τώρα.

Το βασικότερο μειονέκτημα που έχουν τα **AntiVirus** είναι ότι δεν μπορούν να εντοπίσουν επιθέσεις από ιούς που δεν είναι γνωστοί αν και σήμερα έχουν φτάσει στο σημείο τουλάχιστον να διακρίνουν νέες επιθέσεις που στην πραγματικότητα είναι παραλλαγές παλαιοτέρων επιθέσεων. Η αδυναμία τους αυτή οφείλεται στο μηχανισμό με τον οποίο τα AnitVirus εντοπίζουν γενικά τους ιούς και ειδικότερα αυτούς **που βασίζονται σε Buffer Overflow**

Exploits. Ο μηχανισμός αυτός δεν είναι τίποτα παραπάνω από ένα μηχανισμό **Pattern Matching**.

Ο μηχανισμός του **Pattern Matching** δεν κάνει τίποτα παραπάνω από το να προσπαθεί να εντοπίσει μέσα στην String μορφή ενός προγράμματος που παίρνει σαν είσοδο αν το περιέχει String όμοια με τα γνωστά **BOE String** ή άλλων Virus. Όπως είναι κατανοητό αν το String ενός BOE δεν το γνωρίζει το AntiVirus τότε δεν μπορεί να εντοπίσει το BOE αυτό.

3.4.3 Τα NIDS

Τα τελευταία χρόνια έχουν εμφανιστεί στην αγορά τα **NIDS** με σκοπό να εντοπίζουν επιθέσεις εναντίον **ενός ολόκληρου δικτύου** και όχι μόνο σε επίπεδο ενός υπολογιστή. Η βασική φιλοσοφία λειτουργία τους είναι όμοια με αυτή των **AntiVirus** δηλαδή προσπαθούν να κάνουν **Pattern Matching γνωστών επιθέσεων**.

Η βασική τους διαφορά που έχουν από τα **AntiVirus** είναι ότι αντί να προσπαθούν να εντοπίσουν ένα **Buffer Overflow** ή άλλου είδος επίθεση **μέσα σε αρχεία** προσπαθεί να κάνει το ίδιο στα **πακέτα που περνούν μέσα από ένα δίκτυο.** Η δεύτερη διαφορά τους είναι ότι συνήθως **δεν χρησιμοποιείται ο μηχανισμός αντίδρασης που διαθέτουν** όπως στα Anti-Virus, ενώ αντίθετα τα τελευταία **στηρίζουν την κατασταλτική τους συμπεριφορά** σε αυτόν ακριβώς τον μηχανισμό.

Η τεχνολογία των **NIDS** προς το παρόν σαν βασικό σκοπό έχουν την αυτοματοποιημένη παρακολούθηση ενός δικτύου όπως συμβαίνει με μία **κάμερα ασφάλειας** για παράδειγμα σε μία τράπεζα. Ο ρόλος των NIDS συνεπώς είναι να βοηθήσει στην ενίσχυση της ασφάλεια ενός **Πληροφοριακού Συστήματος** που στηρίζεται στην δικτύωση των υπολογιστών που το αποτελούν αφήνοντας **την κατασταλτική αντίδραση** στον Διαχειριστή του δικτύου.

Η λειτουργία των NIDS όπως ειπώθηκε και παραπάνω δεν διαφέρει κατά πολύ από αυτή του Anti-Virus και δεν σταματά την επίθεση όπως αυτό. Στην ορολογία των NIDS η προσπάθεια τους να εντοπίσουν **Patterns** από επιθέσεις όπως αυτή των **Buffer Overflow Exploits** ονομάζεται **Misused Based Detection**. Εκτός όπως από αυτό το μηχανισμό τα NIDS διαθέτουν και ένα μηχανισμό που λέγεται **Anomaly Based Detection**.

Ο μηχανισμός **Anomaly Based Detection** που διαθέτουν τα NIDS είναι ένας μηχανισμός που προέκυψε σαν απάντηση στο πρόβλημα που έχουν και τα **AntiVirus** ότι δηλαδή δεν είναι δυνατόν να εντοπιστεί μία επίθεση αν δεν είναι γνωστό το Pattern του String του **Buffer Overflow Exploit** ή άλλης επίθεσης. Ο **Anomaly Based Detection** μηχανισμός προσπαθεί να εντοπίσει επιθέσεις **πριν αυτές γνωστοποιηθούν** και ενσωματωθούν στο *Misused Based Detection* μηχανισμό. Για να το κάνει αυτό αναλόγως με τη περίπτωση **μοντελοποιείται η φυσιολογική συμπεριφορά των πακέτων** και αν παρατηρηθεί κάποια **ανωμαλία σε αυτά ή στο περιεχόμενο τους** τότε αποστέλλεται σχετική ειδοποίηση στον διαχειριστή του δικτύου.

Περισσότερες λεπτομέρειες για το τα NIDS μπορείται να βρείται στο Technical Report “Snort 2.0&Snort_Preprocessors” που παρουσιάζεται το Snort 2.0 το **γνωστότερο OpenSource NIDS** σήμερα(2003). Τέλος **η λύση για την αντιμετώπιση των Buffer Overflow που υλοποιήθηκε στα πλαίσια αυτή της πτυχιακής** στηρίζει την λειτουργία του στο **Snort 2.0** ενώ προστίθεται σε αυτό σαν **module**. Το module αυτό είναι μία μέθοδος εντοπισμού που στην πράξη υλοποιεί έναν **Anomaly Based Detection μηχανισμό εντοπισμού των Buffer Overflow Exploits**.

Ο σκοπός αυτή της λύσης για το **Buffer Overflow** όπως και ο λόγος ύπαρξης των NIDS γενικότερα, είναι να **βοηθήσει στην καταστολή της εξάπλωσης** των **Buffer Overflow Exploit** μέσα σε ένα δίκτυο **ειδικά όταν αυτά δεν είναι γνωστά** άρα δεν μπορούν να εντοπιστούν από τον **Misused Based Detection** ενός NIDS ή **AntiVirus**.

Λεπτομέρειες για το μηχανισμό εντοπισμού των **Buffer Overflow Exploits** παρουσιάζονται στο ***κεφαλαίο 5*** της πτυχιακής. Πριν από αυτό όμως θα παρουσιαστεί μία ακόμα **μέθοδος κατασταλτικού μηχανισμού των BOE η οποία είναι και η βάση της δικιάς μου υλοποίησης**.

3.4.4 Accurate Buffer Overflow Detection via Abstract Execution of Payload

Η λύση αυτή προτάθηκε στην **RAID 2002** από τους **T.Toth** και **C.Kruegel** και παράδειγμα υλοποίησης αυτής της ιδέας έγινε σαν module του Apache Server. Η βασική ιδέα της λύσης είναι ότι σε κάθε εφαρμογή που παρέχει δικτυακές υπηρεσίες πρέπει να **ενσωματωθεί ένα module που με κάποιο μηχανισμό** θα ελέγχει τα εισερχόμενα πακέτα για να βρει Buffer Overflow Exploits.

Ο μηχανισμός αυτός για να βρει BOE ελέγχει το String που δέχεται σαν είσοδο δηλαδή τα **περιεχόμενα ενός πακέτου(Payload)**. Αν μέσα στο πακέτο βρεθεί ένα BOE τότε αυτό δεν παραδίδεται στην υπόλοιπη εφαρμογή προστατεύοντας την έτσι από τέτοιου είδους επιθέσεις. Για να μπορέσει όμως να λειτουργήσει αυτό ο μηχανισμός θα πρέπει να μπορεί με κάποιο τρόπο να ξεχωρίζει **τα πακέτα που περιέχουν BOE** από τα υπόλοιπα.

Ο τρόπος που ξεχωρίζει τα πακέτα μεταξύ τους βασίζεται στο **μεγάλο βαθμό εκτελεσιμότητας** που έχουν τα πακέτα με BOE σε σχέση με τα υπόλοιπα**. Εκτελεσιμότητα σε ένα πακέτο σημαίνει** ότι το String που αποτελεί το payload ενός πακέτου μπορεί να αντιστοιχεί ολόκληρο ή κομμάτια του **σε αλυσίδες εκτελέσιμων εντολών**. Η έννοια της εκτελεσιμότητας θα εξεταστεί αναλυτικά στο **κεφάλαιο 4**.

Το σημαντικό που πρέπει να γνωρίζει κάποιο για αυτή τη μέθοδο είναι ότι η εκτελεσιμότητα μπορεί να **παρουσιάζεται σε ένα πακέτο τυχαία**. Όμως όταν το πακέτα περιέχει Buffer Overflow Exploit String τότε **η εκτελεσιμότητα παρουσιάζεται σίγουρα**. Για να μπορέσει λοιπόν να ξεχωρίζει η μέθοδος αυτή πότε ένα πακέτο παρουσιάζει εκτελεσιμότητα κατά τύχη και πότε όχι παίρνει σαν είσοδο για κάθε δικτυακή υπηρεσία **μία τιμή κατώφλι(Threshold) και με αυτή** διαχωρίζει τα πακέτα. Η τιμή αυτή **καθορίζεται με στατιστικές μετρήσεις**.

Αν και η μέθοδος αυτή είναι δύσκολο να ενσωματωθεί σε κάθε δυνατή υπηρεσία ενός δικτύου όπως προτείνεται στο paper έχει πολύ καλά αποτελέσματα. Επειδή η μέθοδος αυτή έχει πολύ καλά αποτελέσματα στον εντοπισμό των επιθέσεων Buffer Overflow Exploit η μέθοδος αυτή χρησιμοποιήθηκε από τον συγγραφέα αυτή της πτυχιακής εργασίας για να υλοποίηση μια έκδοση του Snort 2.0 που θα εντοπίζει τέτοιες επιθέσεις χωρίς προηγουμένως αυτές να είναι γνωστές.