Return-to-libc 攻击实验

1 概述

本实验的学习目标是让学生亲身体验缓冲区溢出攻击的一个有趣的变种，这种攻击可以绕过目前 Linux 操作系统中实现的保护方案。利用缓冲区溢出漏洞的常见方法是将恶意的 shellcode 注入到被攻击者的栈上，然后让被攻击的程序跳转到 shellcode。为了防止这类攻击，一些操作系统允许程序将其堆栈设置成不可执行，这样的话，跳转到 shellcode 会导致程序失败。

不幸的是，上述保护方案并非万无一失。存在一种称为 *Return-to-libc* 的缓冲区溢出攻击变种，它不需要可执行堆栈，甚至不使用 shellcode。相反，它使易受攻击的程序跳转到一些现有代码，例如进程内存空间中已加载的 `libc` 库中的 `system()` 函数。

在这个实验中，学生将获得一个具有缓冲区溢出漏洞的程序，他们的任务是开发一个 Return-to-libc 攻击来利用这个漏洞，并最终获得 root 权限。除了攻击之外，学生还将了解 Ubuntu 中实现的一些保护方案，以反击缓冲区溢出攻击。本实验涵盖以下主题：

- 缓冲区溢出漏洞
- 函数调用时的堆栈布局和不可执行堆栈
- Return-to-libc 攻击和返回导向编程（ROP）

参考材料和视频。 可以在以下资源中找到关于 return-to-libc 攻击的详细参考：

- SEED 书籍，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED 网课第 5 节，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

实验环境。 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

教师说明。 教师可以通过选择易受攻击程序中的缓冲区大小值来定制此实验。详见第 2.3节。

2 环境设置

2.1 x86 和 x64 架构的说明

在 x64 机器（64 位）上的 return-to-libc 攻击比在 x86 机器（32 位）上的要困难得多。尽管 SEED Ubuntu 20.04 VM 是一台 64 位机器，我们决定继续使用 32 位程序（x64 与 x86 兼容，因此 32 位程序仍可在 x64 机器上运行）。将来，我们可能会为这个实验引入 64 位版本。因此，在此实验中，当我们使用 `gcc` 编译程序时，我们总是使用`-m32` 标志，这意味着将程序编译成 32 位二进制文件。

2.2 关闭安全机制

你可以使用我们预构建的 Ubuntu 虚拟机来执行实验任务。Ubuntu 和其他 Linux 发行版已经实现了几个安全机制，使得缓冲区溢出攻击变得困难。为了简化我们的攻击，我们需要先禁用它们。

地址空间随机化。 Ubuntu 和其他几个基于 Linux 的系统使用地址空间随机化来随机化堆和堆栈的起始地址，使得猜测确切地址变得困难。猜测地址是缓冲区溢出攻击的关键步骤之一。在此实验中，我们使用以下命令关闭地址空间随机化：

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

堆栈保护方案 StackGuard。 `gcc` 编译器实现了一个名为 *StackGuard* 的安全机制来防止缓冲区溢出。在这种保护存在的情况下，缓冲区溢出攻击无法工作。我们可以通过在编译期间使用*-fno-stack-protector* 选项来禁用此保护。例如，要编译一个禁用 StackGuard 的程序 `example.c`，我们可以这样做：

```
$ gcc -m32 -fno-stack-protector example.c
```

不可执行堆栈。 Ubuntu 曾经允许可执行堆栈，但现在已改变。程序（和共享库）的二进制映像必须声明它们是否需要可执行堆栈，即它们需要在程序头中标记一个字段。内核或动态链接器使用这个标记来决定是否使此运行程序的堆栈可执行或不可执行。这个标记是由 `gcc` 自动完成的，默认情况下，堆栈被设置为不可执行。要改变这一点，请在编译程序时使用以下选项：

```
可执行堆栈
$ gcc -m32 -z execstack  -o test test.c

不可执行堆栈
$ gcc -m32 -z noexecstack  -o test test.c
```

由于本实验的目标是展示不可执行堆栈保护不起作用，你应该在本实验中使用`"-z noexecstack"` 选项来编译你的程序。

配置/bin/sh。 在 Ubuntu 20.04 中，`/bin/sh` 符号链接指向`/bin/dash` shell。`dash` shell 有一个对策，可以阻止自己在 `Set-UID` 进程中执行。如果 `dash` 在 `Set-UID` 进程中执行，它会立即将有效用户 ID

更改为进程的实际用户 ID，实质上放弃了其权限。

由于我们的受害者程序是一个 Set-UID 程序，我们的攻击使用 system() 函数来运行我们选择的命令。这个函数不会直接运行我们的命令，它调用/bin/sh 来运行我们的命令。因此，/bin/dash 在执行我们的命令之前就放弃了 Set-UID 权限，使我们的攻击更加困难。要禁用此保护，我们将/bin/sh 链接到另一个没有这种对策的 shell。我们在 VM 中安装了一个名为 zsh 的 shell 程序。我们使用以下命令将/bin/sh 链接到 zsh：

```
$ sudo ln -sf /bin/zsh /bin/sh
```

应当注意，dash 中实现的对策是可以绕过的。我们将在后续任务中进行。

2.3 易受攻击的程序

Listing 1: 易受攻击的程序（retlib.c）

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

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // 将ebp复制到framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* 为了实验目的打印信息 */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);   ← buffer overflow!

    return 1;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;
```

```
   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}

// 此函数将在可选任务中使用
void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}
```

上述程序具有缓冲区溢出漏洞。它首先从名为 `badfile` 的文件中读取多达 1000 字节的输入。然后，它将输入数据传递给 `bof()` 函数，该函数使用 `strcpy()` 将输入复制到其内部缓冲区。然而，内部缓冲区的大小小于 1000，因此存在潜在的缓冲区溢出漏洞。

这个程序是一个 root 拥有的 `Set-UID` 程序，因此如果普通用户可以利用这个缓冲区溢出漏洞，用户能够获得 root shell。该程序从用户提供的名为 `badfile` 的文件中获取输入，因此，我们可以构造该文件，以便当易受攻击的程序将文件内容复制到其缓冲区时，可以生成 root shell。

编译。 让我们首先编译代码并将其变成 root 拥有的 `Set-UID` 程序。不要忘记用`-fno-stack-protector` 选项（关闭 StackGuard 保护）和`"-z noexecstack"` 选项（打开不可执行堆栈保护）。还应注意，在打开 `Set-UID` 位之前必须先更改所有权，因为所有权更改会导致 `Set-UID` 位被关闭。所有这些命令都包含在提供的 `Makefile` 中。

```
  // 注意: N应替换为教师设置的值
$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

给教师的建议。 为了防止学生使用过去（或在互联网上发布的）的解决方案，教师可以通过要求学生使用不同的 `BUF_SIZE` 值来编译代码，从而改变 `BUF_SIZE` 的值。如果没有`-DBUF_SIZE` 选项，`BUF_SIZE` 将设置为默认值 12（在程序中定义）。当这个值改变时，堆栈的布局将会改变，解决方案也会不同。学生应向他们的教师询问 `N` 的值。`N` 的值可以在提供的 `Makefile` 中设置，`N` 可以从 10 到 800。

3 实验任务

3.1 任务 1：找出 libc 函数的地址

在 Linux 中，当程序运行时，libc 库将被加载到内存中。当内存地址随机化关闭时，对于相同的程序，库总是被加载到相同的内存地址中（对于不同的程序，libc 库的内存地址可能不同）。因此，我们可以使用调试工具，如 gdb，找出 system() 的地址。也就是说，我们可以调试目标程序 retlib。尽管程序是一个 root 拥有的 Set-UID 程序，我们仍然可以调试它，只是权限将被丢弃（即，有效用户 ID 将与真实用户 ID 相同）。在 gdb 中，我们需要输入 run 命令来执行一次目标程序。否则，库代码将不会被加载。我们使用 p 命令（或 print）打印出 system() 和 exit() 函数的地址（我们稍后将需要 exit()）。

```
$ touch badfile
$ gdb -q retlib     ← 使用 ``安静'' 模式
Reading symbols from ./retlib...
(No debugging symbols found in ./retlib)
gdb-peda$ break main
Breakpoint 1 at 0x1327
gdb-peda$ run
......
Breakpoint 1, 0x56556327 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
```

应当注意，即使对于相同的程序，如果我们将其从 Set-UID 程序更改为非 Set-UID 程序，libc 库可能不会被加载到同一位置。因此，当我们调试程序时，需要调试目标 Set-UID 程序，否则，我们得到的地址可能是错误的。

在批处理模式下运行 gdb。 如果你更喜欢在批处理模式下运行 gdb，你可以将 gdb 命令放入一个文件中，然后让 gdb 执行这个文件中的命令：

```
$ cat gdb_command.txt
break main
run
p system
p exit
quit
$ gdb -q -batch -x gdb_command.txt ./retlib
...
Breakpoint 1, 0x56556327 in main ()
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

3.2 任务 2：将 shell 字符串放入内存

我们的攻击策略是跳转到 system() 函数，并使其执行任意命令。由于我们希望获得 shell 提示符，我们希望 system() 函数执行"/bin/sh" 程序。因此，命令字符串"/bin/sh" 必须首先放入内存中，我们必须知道其地址（这个地址需要传递给 system() 函数）。有很多方法可以实现这些目标，我们选择一种使用环境变量的方法。学生也可以使用其他方法。

当我们从 shell 提示符执行程序时，shell 实际上会生成一个子进程来执行程序，所有被 export 的 shell 变量会成为子进程的环境变量。我们可以通过这个方法将一个任意字符串放入子进程的内存中。让我们定义一个新的 shell 变量 MYSHELL，并让它包含字符串"/bin/sh"。从以下命令中，我们可以验证字符串进入了子进程，并由在子进程中运行的 env 命令打印出来。

```
$ export MYSHELL=/bin/sh
$ env | grep MYSHELL
MYSHELL=/bin/sh
```

我们将使用这个变量的地址作为 system() 调用的参数。这个变量在内存中的位置可以使用以下程序找到：

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

将上述代码编译成名为 prtenv 的二进制文件。如果关闭了地址随机化，你会发现打印出的是相同的地址。当你在同一终端中运行易受攻击的程序 retlib 时，环境变量的地址将是相同的（见下面的特别说明）。你可以通过将上述代码放入 retlib.c 中来验证这一点。然而，程序名称的长度确实有影响。这就是我们选择 6 个字符作为程序名称 prtenv 以匹配 retlib 的原因。

说明。 当你编译上述程序时，应使用-m32 标志，因为二进制代码 prtenv 是为 32 位机器准备的，而不是 64 位机器。易受攻击的程序 retlib 是一个 32 位二进制文件，所以如果 prtenv 是 64 位的，环境变量的地址将会不同。

3.3 任务 3：发起攻击

我们准备创建 badfile 的内容。由于内容涉及二进制数据（例如，libc 函数的地址），我们可以使用 Python 进行构造。我们提供了以下代码框架，关键部分留待你填写。

```
#!/usr/bin/env python3
import sys

# 用非零值填充
content = bytearray(0xaa for i in range(300))

X = 0
```

```
sh_addr = 0x00000000       #  "/bin/sh" 字符串的地址
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # system() 函数的地址
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # exit() 函数的地址
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# 保存内容到文件
with open("badfile", "wb") as f:
  f.write(content)
```

你需要找出三个地址和 X、Y 和 Z 的值。如果你的值不正确，你的攻击可能不会成功。在你的报告中，你需要描述是如何决定 X、Y 和 Z 的值的，并展示你的推理过程。如果你使用试错方法，展示你的尝试过程。

关于 gdb 的说明。 如果你使用 gdb 来找出 X、Y 和 Z 的值，应当注意到 Ubuntu 20.04 中的 gdb 行为与 Ubuntu 16.04 中的略有不同。在我们设置断点在函数 bof 后，当 gdb 在 bof() 函数内部停止时，它会在 ebp 寄存器设置指向当前堆栈帧之前停止，所以如果我们在这里打印出 ebp 的值，我们将得到调用者的 ebp 值，而不是 bof 的 ebp。我们需要输入 next 执行一些指令后，在 ebp 寄存器指向 bof() 函数的栈帧后停止。SEED 书（第 1 版）是基于 Ubuntu 16.04，没有这个 next 步骤。

攻击变种 1： exit() 函数真的必要吗？请尝试在 badfile 中不包括这个函数的地址，再次运行你的攻击，报告并解释你的观察结果。

攻击变种 2： 在你的攻击成功后，将 retlib 的文件名更改为不同的名称，确保新文件名的长度不同。例如，你可以将其更改为 newretlib。重复攻击（不改变 badfile 的内容）。你的攻击会成功吗？如果没有成功，解释原因。

3.4 任务 4：击败 Shell 的对策

本任务的目的是在启用了 shell 的对策后发起 return-to-libc 攻击。在执行任务 1 至 3 之前，我们将/bin/sh 重新链接到了/bin/zsh，而不是/bin/dash（原始设置）。这是因为一些 shell 程序，如 dash 和 bash，在 Set-UID 进程中执行时会自动放弃权限。在此任务中，我们希望击败此类对策，即即使/bin/sh 仍然指向/bin/dash，我们也希望获得 root shell。首先，我们将符号链接改回：

```
$ sudo ln -sf /bin/dash /bin/sh
```

尽管 dash 和 bash 都会放弃 Set-UID 权限，但如果-p 选项被调用，权限将不被放弃。当我们返回到 system 函数时，这个函数会调用/bin/sh，但它不使用-p 选项。因此，目标程序的 Set-UID 权

限将被放弃。如果有一个函数允许我们直接执行"/bin/bash -p", 而不是像 system 那样，我们就可以获得 root 权限。

很多 libc 函数都可以做到这一点，例如 exec() 系列函数，包括 execl()、execle()、execv() 等。让我们看看 execv() 函数。

```
int execv(const char *pathname, char *const argv[]);
```

这个函数接受两个参数，一个是命令的地址，第二个是命令的参数数组的地址。例如，如果我们想使用 execv 调用"/bin/bash -p", 我们需要设置如下：

```
pathname = address of "/bin/bash"
argv[0]  = address of "/bin/bash"
argv[1]  = address of "-p"
argv[2]  = NULL (i.e., 4 bytes of zero).
```

从前面的任务中，我们可以轻松获得字符串的地址。因此，如果我们能在堆栈上构造 argv[] 数组，并获取其地址，我们就能进行 return-to-libc 攻击，但这一次，我们将返回到 execv() 函数。

这里有一个难点。argv[2] 的值必须是零（一个整数零，四个字节）。如果我们在输入中放入四个零，strcpy() 将在第一个零处终止，任何在它之后的内容都不会被复制到 bof() 函数的缓冲区中。这似乎是一个难题，但请记住，你的输入中的所有内容已经在堆栈上，它们在 main() 函数的缓冲区中。获取这个缓冲区的地址并不难。为了简化任务，我们已让易受攻击的程序为你打印出该地址。

就像任务 3 一样，你需要构造你的输入，以便当 bof() 函数返回时，它返回到 execv()。后者从堆栈中获取"/bin/bash" 字符串的地址和 argv[] 数组的地址。你需要在堆栈上准备好一切，以便当 execv() 被执行时，它可以执行"/bin/bash -p"，从而获得 root shell。在你的报告中，请描述你是如何构造输入的。

3.5 任务 5（可选）：返回导向编程

解决任务 4 中的问题有很多方法。另一种方法是在调用 system() 之前调用 setuid(0)。setuid(0) 调用将真实用户 ID 和有效用户 ID 都设置为 0，将进程转变为非 Set-UID 进程（它仍然具有 root 权限）。这种方法要求我们将两个函数链接在一起。这种方法被推广为链接多个函数，并且进一步推广为链接多段代码。这就是返回导向编程（ROP）。

使用 ROP 解决任务 4 中的问题相当复杂，它超出了本实验的范围。然而，我们希望给学生一个 ROP 的体验，让他们处理 ROP 的一个特例。在 retlib.c 程序中，有一个名为 foo() 的函数，程序中从未被调用。该函数是为本任务准备的。你的任务是利用程序中的缓冲区溢出问题，使得程序从 bof() 函数返回时，调用 foo() 10 次，然后给你 root shell。在你的实验报告中，你需要描述你的输入是如何构造的。结果将如下所示。

```
$ ./retlib
...
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
```

```
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
bash-5.0#   ← Got root shell!
```

引导。 让我们回顾一下我们在任务 3 中所做的。我们在堆栈上构造数据，使得当程序从 `bof()` 返回时，它跳转到 `system()` 函数，并且当 `system()` 返回时，程序跳转到 `exit()` 函数。我们将在这里使用类似的策略。我们不会跳转到 `system()` 和 `exit()`，而是在堆栈上构造数据，使得当程序从 `bof` 返回时，它返回到 `foo`；当 `foo` 返回时，它返回到另一个 `foo`。这个过程重复 10 次。当第 10 个 `foo` 返回时，它返回到 `execv()` 函数，给我们 root shell。

进一步参考。 我们在本任务中所做的只是 ROP 的一个特例。你可能已经注意到 `foo()` 函数不接受任何参数。否则，调用它 10 次将变得更加复杂。通用的 ROP 技术允许你按顺序调用任何数量的函数，允许每个函数有多个参数。SEED 书籍（第 2 版）提供了如何使用通用 ROP 技术解决任务 4 中问题的详细说明。它涉及调用 `sprintf()` 四次，然后调用 `setuid(0)`，再调用 `system("/bin/sh")` 给我们 root shell。这种方法相当复杂，SEED 书籍中用了 15 页的内容来解释。

4 指南：理解函数调用机制

4.1 理解堆栈布局

要了解如何进行 Return-to-libc 攻击，我们需要理解堆栈的工作原理。我们使用一个小型 C 程序来理解函数调用对堆栈的影响。更详细的解释可以在 SEED 书籍和 SEED 课程中找到。

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
  printf("Hello world: %d\n", x);
}

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

我们可以使用"`gcc -m32 -S foobar.c`" 将这个程序编译成汇编代码。生成的文件 `foobar.s` 将如下所示：

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

4.2 调用和进入 foo()

让我们关注调用 foo() 时的堆栈。我们可以忽略之前的堆栈。请注意，本解释中使用的是行号而不是指令地址。

- **第 28-29 行:**：这两个语句将值 1，即 foo() 的参数，推入堆栈。这个操作使%esp 增加 4。这两个语句之后的堆栈如图 1(a) 所示。
- **第 30 行:** call foo：该语句将紧随 call 语句之后的下一条指令的地址推入堆栈（即返回地址），然后跳转到 foo() 的代码。当前堆栈如图 1(b) 所示。
- **第 9-10 行**：函数 foo() 的第一行将%ebp 推入堆栈，以保存先前的栈帧指针。第二行让%ebp 指向当前栈帧。当前堆栈如图 1(c) 所示。
- **第 11 行:** subl $8, %esp：堆栈指针被修改，以便为局部变量和传递给 printf 的两个参数分配空间（共 8 字节）。由于函数 foo 中没有局部变量，这 8 字节仅用于参数传递。见图 1(d)。

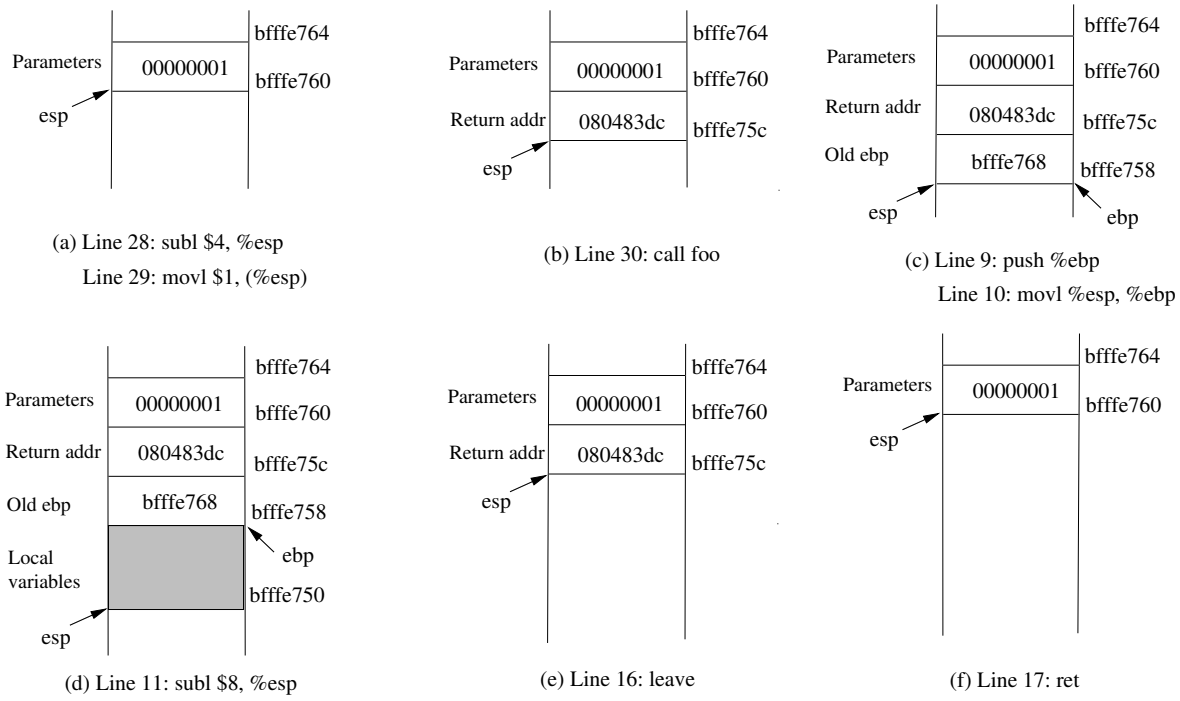

图 1: 进入和离开 `foo()`

4.3 离开 `foo()`

现在控制权已经传递给函数 `foo()`。让我们看看函数返回时堆栈会发生什么。

- **第 16 行：`leave`**：这条指令其实执行了两个指令（它在早期 x86 版本中是一个宏，但后来被制作成了指令）：

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

  第一条语句释放为函数分配的堆栈空间；第二条语句恢复先前的框架指针。当前堆栈如图 1(e) 所示。

- **第 17 行：`ret`**：这条指令从堆栈中弹出返回地址，然后跳转到返回地址。当前堆栈如图 1(f) 所示。

- **第 32 行：`addl $4, %esp`**：进一步释放为 `foo` 分配的内存。如你所见，堆栈现在的状态与进入函数 `foo` 之前完全相同（即，第 28 行之前）。

5 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码

不加以解释不会获得学分。