Pipe WriteUp - EkoParty pre-CTF 2020

Lautaro Pinilla
lpinilla@itba.edu.ar

13 de Septiembre de 2020

Contenidos

1 Introducción

En el siguiente documento se va a presentar una forma de resolver el ejercicio "Pipe" de la categoría de "reverse" presentado en la pre-CTF organizada para la EkoParty de manera virtual en el mes de Agosto. Para resolver el ejercicio, se necesita el conocimiento previo adquirido en Arquitectura de Computadoras y conocer los distintos ataques a funciones de encriptación vistos en la materia Criptografía y Seguridad.

Sin embargo, de haber un lector curioso por los temas, le recomiendo que investigue por su cuenta los conceptos que no conozca para sacar el máximo provecho de este escrito.

1.1 Consigna

El ejercicio está compuesto por un archivo binario y un texto que dice

```
1 My secrets are safe after running this tool:
2 cat secrets.txt | ./Pipe
3 d740a5dc607f78fbffe520efc7caebd2137940ddb26c30c2fd37ed743b
4 77038d326a9c7e7e80
5 You could need this:
6 MD5(secrets.txt)=080d5caaed95af9ab072c41de3a73c24
```

Mirando el comando, se puede deducir que el programa Pipe realizar algún método de encriptación y que el objetivo del desafío es recuperar el mensaje original de *secrets.txt* en base a el mensaje cifrado. Además, se nos proveé un Hash MD5 sobre el archivo que nos podría llegar a servir como método de validación del mensaje a recuperar.

2 Primer análisis

El primer objetivo para resolver el desafío es reconocer con qué elementos debemos trabajar, de esta manera vamos a poder concentrar nuestro tiempo y nuestros esfuerzos de mejor manera. Sin embargo, no debemos olvidarnos de probar cosas simples que tal vez nos orienten mejor hacia el resultado. Debido a las características de los hashes, podemos descartar analizar el hash MD5 ya que no nos va a brindar información sobre el contenido del mensaje original[1].

Por lo tanto, vamos a centrar todo el foco en hacer ingeniería inversa sobre el archivo binario. Lo primero a realizar entonces es ver qué tipo de archivo es, que podemos ver utilizando el comando *file*:

```
1     $ELF 32-bit LSB shared object, Intel 80386, version 1
         (SYSV), dynamically linked, interpreter
         /lib/ld-linux.so.2, for GNU/Linux 2.6.24,
         BuildID[sha1]=cd439ea515ecfbb1280cfacd93e0703ed9c4d38c,
         not stripped
```

De la siguiente salida podemos ver un par de cosas interesantes, en primer lugar, es un archivo ejecutable en Linux de 32 bits, específicamente compilado para el procesador 80386. Segundo, está linkeado dinámicamente, por lo que depende del SO para resolver las dependencias que tenga y por último, nos dice que es *not stripped*, esto significa que no se extrajeron los símbolos de debug (como por ejemplo, los nombres de las funciones), lo que va a facilitar mucho la hora de entender el código.

3 Analizando el archivo ejecutable

Hay varias formas en las que podemos analizar el archivo que nos dieron, del cual lo único que pudimos deducir es que debe ser algún programa de encriptación. Para analizar el archivo, se puede hacer una análisis estático o dinámico.

El análisis estático involucra mirar el código assembler del programa e interpretar la lógica del programa, mientras que el análisis dinámico involucra correr el programa (con alguna herramienta) y ver cómo evoluciona el flujo del programa dado cierto input. Personalmente, prefiero realizar un análisis estático y correr el

[1] Una de las propiedades que cumplen los hashes es la resistencia a pre-imágenes, esto significa que, dado el hash de un archivo, uno no debería poder conocer el contenido de este. Si bien MD5 ya se considera no segura, vamos a descartar el crackeo del hash como solución.

programa sólo cuando estoy seguro de lo que hace.

Para analizar el archivo, voy a utilizar el programa ghidra[2] que ofrece tanto un decompilador como un "reconstructor" de código C. Por lo que, el programa puede recrear en cierta parte el código C a partir de las instrucciones de assembler. Esta funcionalidad no es del todo confiable ya que siempre requiere intervención personal para evaluar ciertos aspectos, pero aún así es una muy buena base para comenzar.

Para la siguiente parte es muy importante el conocimiento adquirido en la materia Arquitectura de Computadoras. Si bien el programa nos permite tener una aproximación a un código C del binario, debemos recordar que el compilador agrega intrucciones de más como el mecanismo de *canary*[3] y está en nosotros reconocer estas cosas para poder descartarlas al analizar el programa.

En la siguiente foto podemos ver la interpretación de C de la función main:

Figura 1: Función main reconstruida por Ghidra

```
undefined4 main(void)

{
  char cVar1;
  uint uVar2;
  undefined4 uVar3;
  int in_GS_OFFSET;
  int local_98;
  char local_94 [128];
  int local_14;

  local_14 = *(int *)(in_GS_OFFSET + 0x14);
  srand((uint)main);
  fgets(local_94,0x80,stdin);
  local_98 = 0;
  while ((local_94[local_98] != '\0' && (local_94[local_98] != '\n'))) {
    cVar1 = local_94[local_98];
    uVar2 = rand();
    printf("%02x",(int)cVar1 ^ uVar2 & 0xff);
    local_98 = local_98 + 1;
  }
  putchar(10);
  uVar3 = 0;
  if (local_14 != *(int *)(in_GS_OFFSET + 0x14)) {
    uVar3 = __stack_chk_fail_local();
  }
  return uVar3;
}
```

[2]Ghidra es un programa Open Source hecho por la NSA para realizar ingeniería inversa (`https://ghidra-sre.org/`)

[3]El Canary es un valor que se guarda al comienzo de la ejecución y se verifica al terminarla. Si por algún motivo el valor del canary no es correcto, se asume que un atacante intentó sobreescribir el stack y por lo tanto, se corta la ejecución del programa.

Como podemos ver, si bien el programa realiza una buena interpretación inicial, hay algunas cosas en las que tuvo que interpretar como pudo las instrucciones. Un ejemplo sería el uso del tipo de dato undefined4 que haría referencia a un valor de 4 bytes que, siendo un ejecutable de 32 bits, lo más probable es que sea un int. Además, podemos empezar a ver los mecanismos mencionados anteriormente.

Si prestamos atención a la variable llamada "local_14", podemos ver que se utiliza para guardar un valor antes de ejecutar el programa y al finalizar, se verifica si este valor se mantuvo constante o se llama a la función *"__stack_chk_fail_local"*. Esto nos es ni más ni menos que el valor del canary para evitar (o complicar) el buffer-overflow.

3.1 Analizando si hay Buffer Overflow

El Buffer Overflow es una de las técnicas más conocidas que consiste en sobrecargar algún buffer de entrada para poder controlar algún valor del programa (como el valor de retorno).

Sabemos que el programa recibe algo por input, lo modifica y luego imprime el mensaje cifrado. Algo que podemos ver es que en la llamada a la función *fgets* le pasan como argumentos: un buffer de 128 caracteres, un límite de lectura de valor 128 (0x80 en hexa) y el file descriptor de stdin. Al tener un límite de lectura, se descarta el buffer overflow para resolver el ejercicio, ya que de introducir más caracteres que los que el programa espera, la función solo agarra hasta 128 e ignora el resto.

3.2 Analizando la función de Encripción

Luego de cargar el input en el buffer, el loop de while se encarga de realizar la encripción, esta es la función más importante para nosotros ya que vamos a poder saber exáctamente qué es lo que ocurre con el input.

Primero vamos a reescribir las variables para que sea más fácil de leer:

Figura 2: Función main con variables renombradas

```
4  undefined4 main(void)
5
6  {
7    char curr_character;
8    uint randVal;
9    undefined4 uVar1;
10   int in_GS_OFFSET;
11   int index;
12   char input_buffer [128];
13   int canary;
14
15   canary = *(int *)(in_GS_OFFSET + 0x14);
16   srand((uint)main);
17   fgets(input_buffer,0x80,stdin);
18   index = 0;
19   while ((input_buffer[index] != '\0' && (input_buffer[index] != '\n'))) {
20     curr_character = input_buffer[index];
21     randVal = rand();
22     printf("%02x",(int)curr_character ^ randVal & 0xff);
23     index = index + 1;
24   }
25   putchar(10);
26   uVar1 = 0;
27   if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
28     uVar1 = __stack_chk_fail_local();
29   }
30   return uVar1;
31 }
```

Podemos ver que el algoritmo de encripción agarra un valor al azar (aplicando una máscara para obtener sus últimos 2 bytes) y hace un xor con cada caracter de la entrada, les suena familiar?

$$E(m) = m \oplus k \tag{1}$$

Este es el mismo esquema que un OTP[4]. Si recuerdan en Criptografía, este esquema nos permite tener un secreto perfecto siempre y cuando la clave que se utilice sea de la misma longitud que la clave (que en este caso lo es) y que la clave sea **aleatoria**.

Ahora lo único que queda por ver es si la llave que se genera es realmente aleatoria. Para eso, debemos ver cómo se inicializó el *seed* o valor semilla del random, en C se suele hacer con la función *srand*. Ahora es donde debemos recordar las clases de Programación Imperativa en donde el profesor nos decía que siempre que utilicemos srand, debemos inicializarlo de la siguiente manera: *srand(time(NULL))*.

[4]El One Time Pad es una técnica de encriptación que cumple con la propiedad de que asegura secreto perfecto, lo que nos indica que no podemos "crackear" el mensaje desencriptado y recuperar el mensaje original.

4 Primer descubrimiento

Recapitulando hasta ahora, tenemos un programa que agarra hasta 128 caracteres de entrada estándar y aplica la función de encripción enunciada arriba pero vemos que no se hace un uso correcto de la inicialización de la semilla, por lo que ya no podemos garantizar que la llave que se usa para desencriptar sea aleatoria y por lo tanto, no podemos asegurar de que el sistema de encripción sea seguro.

Además, como pudimos ver en el código, el output del programa es el valor hexadecimal (en 2 posiciones) del mensaje encriptado, por lo que, si dividimos la longitud del output encriptado de la consigna por 2, podemos obtener la longitud del mensaje original.

Para entender un poco más el problema, repasemos por qué tenemos que inicializar al random con una semilla. Imaginemos que la función random tiene por adentro un array de integers predefinidos y al pedirle un valor aleatorio con la función *rand*, es devolver secuencialmente un valor de este array. Se habrán dado cuenta de que esto es determinístico. Entonces, uno se podría preguntar, ¿por qué se utiliza? Porque se varía el valor semilla. El valor semilla sería un offset a partir de donde leemos el array de ints por lo que, si agarramos un offset al azar, no importa que los números estén "prefijados", siempre van a ser aleatorios para nosotros. En este caso, nosotros estaríamos arrancando siempre desde el mismo offset, entonces si encontráramos ese offset, cada llamada a rand que hagamos nos va a dar el mismo valor que le dio a rand a la hora de encriptar.

4.1 Descubriendo el offset

Podemos ver que el valor que se le pasa como semilla es el valor de la función main, que significaría que es la dirección de memoria donde se encuentra main. Uno podría pensar que ahora simplemente debemos agarrar la dirección de memoria de main y ponerla en el *srand* para resolver el ejercicio. Pero hay algo que no estamos teniendo en cuenta que se llama ASLR (Address Space Layout Randomization).

4.1.1 Entendiendo ASLR

Esta es una técnica que se utiliza para evitar algunos tipos de ataques como el Buffer Overflow. Consiste en cargar la dirección base del programa en un lugar al azar. De esta forma, si un atacante quisiera cambiar el flujo del programa e ir una función o parte de una función en particular (pisando el Instruction Pointer), no va a poder ya que la próxima vez que se ejecute el programa, este va a estar

puesto sobre otra dirección base y por lo tanto, el valor al que el atacante quería ir ya no es el que quería. Sin embargo, las posiciones relativas de las funciones si se mantienen.

5 Segundo descubrimiento

Vimos que se utiliza un esquema de OTP para encriptar el mensaje pero la llave no se genera de forma aleatoria (es más, la podríamos encontrar). Por lo tanto, si nosotros pudiéramos encontrar el valor semilla (la dirección de main) que tenía el programa cuando se ejecutó, podemos aprovecharnos de la propiedad de "anulación" del operador xor[5] y así recuperar el mensaje original de la siguiente manera:

$$E(m) \oplus k = m \oplus k \oplus k = m \tag{2}$$

Por lo tanto, podríamos hacer un programa análogo a Pipe pero inicializando la semilla en el valor correcto. De esta manera, podríamos recuperar el mensaje original. Cabe destacar que también podríamos correr el programa Pipe tomando como entrada el mensaje encriptado (la salida del mismo programa), pero las posibilidades de que el programe se cargue con el mismo valor que había antes sería la probabilidad de tener esos 8 valores hexadecimales fijos (direcciones de 32 bits) de los posibles 16^8 combinaciones (sería más probable que nos caiga un rayo a que se cargue en el mismo valor)[6]. Por lo tanto, una solución por fuerza bruta parece inviable por la necesidad de tiempo de ejecución necesario. Sin embargo, realizando algunas observaciones, podemos realizar algunas optimizaciones.

5.1 Optimizaciones

Supongamos que la dirección de main está en el valor `0x00010251`, tal vez en la próxima corrida la posición sea `0x00040251` (ya que como mencioné antes, ASLR cambia el valor **base**). Entonces podemos ver que si bien la dirección base es aleatoria, los últimos 3 bytes en nuestro ejemplo permanecieron igual (que en el desafío también ocurre), redujendo el espacio de búsqueda de 16^8 a 16^5, un espacio de búsqueda sobre el que si podemos trabajar. Además, podríamos ir de forma descendiente, arrancando del valor `0xfffff251` ya que es muy poco probable que el programa se cargue en los primeros espacios de memoria.

[5]El operador xor no cumple con la propiedad de impotencia, que para lo que nos interesa significa que si aplicamos a xor a, obtenemos 0

[6]La probabilida de que caiga un rayo es de 1/3 millones $\sim 16^5$, mientras que la probabilidad para que justo toque la dirección deseada es de 1/4 billones $\sim 16^8$

6 Exploit

Nuestro objetivo ahora es crear un programa (compilandolo para 32 bits) que haga fuerza bruta sobre el valor de las direcciones de memoria e ir realizando la operación xor contra el mensaje cifrado para obtener el mensaje original. Acá podríamos tener un elemento de la consigna que teníamos olvidado, el hash. Podríamos calcular el hash de cada mensaje que desencriptamos y si es igual al que nos dieron, encontramos el mensaje original.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <openssl/md5.h>
5
6 #define MESSAGE_LEN 38
7
8 //funcion para limpiar los buffers
9 void clean_buffers(char * decoded, unsigned char * digest, char
      * md5string);
10 //funcion para decodificar el mensaje
11 void decode_message(int * coded_hex, char * decoded);
12 //funcion para calcular el hash md5
13 void calculate_md5(MD5_CTX * mdContext, char * decoded,
      unsigned char * digest, char * md5string);
14
15 int main(void){
16     //mensaje encriptado
17     char * coded[MESSAGE_LEN] = {"d7", "40", "a5", "dc", "60",
          "7f", "78", "fb", "ff", "e5", "20", "ef", "c7", "ca",
          "eb", "d2", "13", "79", "40", "dd", "b2", "6c", "30",
          "c2", "fd", "37", "ed", "74", "3b", "77", "03", "8d",
          "32", "6a", "9c", "7e", "7e", "80"};
18     //hash md5
19     char * expected_md5 = "080d5caaed95af9ab072c41de3a73c24";
20     //buffer para guardar el mensaje desencriptado final
21     char decoded[MESSAGE_LEN];
22     //mensaje encriptado como int
23     int coded_hex[MESSAGE_LEN];
24     //buffer para que calculemos el hash
25     unsigned char digest[16];
26     //buffer para convertir el hash a texto
27     char md5string[33];
28     //limpiando los buffers
29     clean_buffers(decoded, digest, md5string);
30     //contexto para calcular el hash md5
31     MD5_CTX mdContext;
32     //convertimos el valor hexadecimal en string a un int
```

```
33      for(int i = 0; i < MESSAGE_LEN; i++){
34          coded_hex[i] = (int)strtol(coded[i], NULL, 16);
35      }
36      //offset que agarramos con analisis estatico
37      uint main_offset = 0x66c;
38      //bruteforce
39      for(uint i = 0xfffff000 + main_offset;
40              i > 0x00001000 + main_offset; i-=4096){
41          //inicializando srand
42          srand((uint)i);
43          decode_message(coded_hex, decoded);
44          calculate_md5(&mdContext, decoded, digest, md5string);
45          if (strcmp((const char *)md5string, expected_md5) == 0){
46              printf("FOUND!!!\n");
47              printf("%s\n", decoded);
48              return 0;
49          }
50      }
51  }
52
53  void calculate_md5(MD5_CTX * mdContext, char * decoded,
54          unsigned char * digest, char * md5string){
55      //calculamos el hash md5
56      MD5_Init(mdContext);
57      MD5_Update(mdContext, decoded, MESSAGE_LEN);
58      MD5_Final(digest, mdContext);
59      //convertimos el hash a texto
60      for(int i = 0; i < 16; i++){
61          sprintf(&md5string[i*2], "%02x", (unsigned int)
                digest[i]);
62      }
63  }
64
65  void decode_message(int * coded_hex, char * decoded){
66      //aplicamos la misma funcion que encripcion
67      for(int j = 0; j < MESSAGE_LEN; j++){
68          decoded[j] = (char) ((int)coded_hex[j] ^ (rand() &
                0xff));
69      }
70  }
71
72  void clean_buffers(char * decoded,
73          unsigned char * digest, char * md5string){
74      memset(decoded, 0, MESSAGE_LEN * sizeof(char));
75      memset(digest, 0, 16 * sizeof(char));
76      memset(md5string, 0, 33 * sizeof(char));
77  }
```

exploit.c

Para compilar, debemos ejecutar el comando:

```
1   $ gcc -Wall -Werror -m32 -o exploit exploit.c -lcrypto
```

El flag *m32* es para indicar la arquitectura y el flag *lcrypto* es para incluir las librerías criptográficas.

Finalmente obtenemos:

```
1   $ ./exploit
2   $ FOUND!!!
3   $ The flag is EKO{bullshit_PIE_over_x86}
```

PIE hace referencia a Position Indenpendent Code, dejo como tarea a la persona que lee buscar de qué se trata.

7 Palabras finales

Pudimos ver cómo un simple desafío de pocas líneas nos llevó a recordar varios conceptos visto en algunas materias. Está bueno notar como, para resolver el ejercicio, no se necesita saber de una cosa específica sino de ir combinando varios conceptos que fuimos aprendiendo en nuestro trayecto. En este caso particular, tuvimos que combinar los conocimientos de:

- Assembler, para poder interpretar el programa (que no siempre se puede reconstruir tan simplemente).
- Programación en C, para poder construir un programa que resuelva el problema
- Arquitecturas de Computadoras, que nos permite interpretar el código y entender con qué elementos estamos trabajando (la solución propuesta se hacía inviable si las direcciones de memoria eran de 64 bits).
- Criptografía, que nos permitió reconocer la función de encripción y saber que había que poner el foco en la generación de la llave. Además de proveernos la forma de atacar a esa función para conocer el mensaje original.

También quiero destacar el hecho de que, para solucionar el desafío, no hacía falta correr el programa que nos dieron. Solo haciendo análisis estático pudimos ver qué hace y a partir de eso, construir una solución para el problema. Si bien el análisis dinámico también es muy útil, tal vez en algunos casos no se pueda ejecutar el programa, como podría ser el análisis de un nuevo malware (aunque también hay medidas para realizar un aislamiento de procesos). Además, por criterio personal, aunque la competencia tenga buen renombre, no quisiera arriesgar a mi computadora ejecutando cualquier archivo sin saber qué es o qué hace.

Por último, espero que este Writeup les haya gustado y que les haya despertado la curiosidad en alguno de los temas provistos. Estos ejercicios también sirven para poder darse cuenta de cuanto fuimos aprendiendo durante la carrera, que a veces no nos damos cuenta enfocándonos solamente en el cuatrimestre actual.