Ingeniería Inversa

SL. Práctica 2

Diego Sanz Fuertes | 825015

6 de Noviembre de 2023

Índice

Resumen ejecutivo

Esta práctica consistía en modificar un programa proporcionado con el fin de que funcionase sin necesitar una llave hardware sin tener acceso al código fuente.

A partir del archivo proporcionado, se ha obtenido información sobre el tipo de ejecutable (ELF x86-64 para linux), las cadenas que contiene (incluyendo la version del compilador y menos cadenas de las esperadas).

Se ha ejecutado el archivo para comprobar su funcionamiento y tras mostrar una de las cadenas encontradas muestra un mensaje de error sin encontrar, indicando posible ofuscación de las cadenas. También se ha ejecutado con otras herramientas como `ltrace` para obtener más información sobre las funciones externas que llama.

Tras añadir los permisos necesarios descubiertos con `ltrace` se ejecutó de nuevo y el mensaje de error fue diferente. Se utilizó `ltrace` otra vez y esta vez apareció la llamada del sistema `ioperm`, indicando que hace algún tipo de operación de entrada salida en un puerto, siendo ese un puerto paralelo.

Al no poder avanzar más, se procedió a explorar y analizar el archivo ejecutable con varios desensambladores, utilizando ghidra principalmente. Una vez desensamblado y pseudo-decompilado el archivo se procede a renombrar variables y funciones siguiendo la lógica del programa hasta tener una idea general de como funciona.

También se explica como funciona la encriptación de las cadenas, la interacción con el puerto paralelo y el programa en general.

Por último se ha modificado el archivo original, eliminando las instrucciones `ioperm`, `in` y `out`, invirtiendo o modificando los saltos condicionales necesarios e ignorando la función de comprobación de permisos ya no necesaria.

El resultado obtenido es un archivo ejecutable el cuál no necesita ni permisos ni llave hardware para ser utilizado.

1. Introducción

En esta práctica se pretende modificar una aplicación protegida para que pueda ser ejecutada sin su clave hardware. No se conoce el sistema operativo ni la arquitectura para los que fue desarrollada.

El objetivo es modificar el ejecutable binario para que pueda ser ejecutado sin problemas mediante varias técnicas de ingeniería inversa.

2. Herramientas utilizadas

Para la realización de esta práctica se han utilizado las siguientes herramientas:

- Desensambladores:
  - Ghidra
    - Script Noop This S**t
  - Binary ninja demo
  - IDA Free
  - Radare / iaito
  - Cutter / Rizin
- Otras herramientas:
  - file
  - strings
  - ltrace
  - lurk (alternativa a strace)
  - Linux man pages online

Las herramientas utilizadas para la realización de la memoria han sido Obsidian y pandoc.

3. Paso 1: Análisis preliminar del binario

Antes de poder ser ejecutado, se necesita conocer la plataforma para la que el binario fue compilado. La manera mas sencilla para comprobar el tipo de un archivo es mediante la herramienta `file`:

```
1 $ file legado_original
2
3 legado_original: ELF 64-bit LSB pie executable,
4 x86-64, version 1 (SYSV), dynamically linked,
5 interpreter /lib64/ld-linux-x86-64.so.2,
6 for GNU/Linux 3.2.0,
7 BuildID[sha1]=9ac025a5231d0f6eb4e0e43922632967340fe7a5,
8 stripped
```

Gracias a la salida de esta herramienta se puede observar que, entre otra información, es un archivo ELF (Executable and Linking Format), el formato de los binarios estándar en Linux. También se indica que la arquitectura para la que fue compilado es x86-64, la más común en los ordenadores personales.

Otra herramienta con la que se puede obtener información a partir de un binario es `strings`, que lista las cadenas de caracteres que se encuentran en el archivo. Algunas de las más relevantes han sido:

```
1 $ strings legado_original --encoding S # utf-8
2
3 ...
4 ls -l %s
5 Stocks v 3.03
6 clear
7 Opción:
8 GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
9 ...
```

Entre las cadenas se encuentran nombres de llamadas al sistema, funciones internas de linux y los nombres de las secciones del ELF. Las cadenas mas relevantes ofrecen información que ha muy útil en los siguientes pasos.

4. Paso 2: Ejecución del programa

Una vez obtenida la información sobre el programa, se puede intentar ejecutar el binario:

```
1 $ ./legado_original
2 Stocks v 3.03
3 Error de permisos de ejecución
```

Al ejecutarlo se muestra la cadena previamente descubierta `Stocks v 3.03` y una cadena desconocida `Error de permisos de ejecución` por lo que se puede deducir que existe ofuscación en el programa. El código de retorno ha sido -1 por lo que la ejecución no ha terminado satisfactoriamente, como era esperable. También se ha intentado ejecutar como usuario root siguiendo el mensaje de error, obteniendo el mismo resultado. Como ultima prueba de ejecución, se le han otorgado todos los permisos mediante `chmod 777 legado_original` sin éxito.

También se ha ejecutado el programa a través de ltrace para obtener más información sobre las llamadas a librerías que se realizan. En el siguiente fragmento de la salida se puede observar que escribe la cadena que se puede ver al ejecutar (“Stocks v 3.03”), se llama a popen que ejecuta el comando `ls -l ./legado_original", "r` y pone su resultado en un stream el cual es leído por `fgets` devolviendo el resultado del `ls`. Se comprueba si tiene los permisos `rws` (sticky bit) y al no encontrarlo parece que imprime un error carácter a carácter, lo que puede indicar el porqué no aparece esa cadena en `strings`.

```
$ ltrace ./legado_original
puts("Stocks v 3.03"Stocks v 3.03
)                       = 14
__sprintf_chk(0x7ffd58369f80, 1, 1024, 0x559828c01624) = 23
popen("ls -l ./legado_original", "r")     = 0x55982a6156b0
fgets( [no return] ...
--- SIGCHLD (Child exited) ---
[... fgets resumed] "-rwxrwxrwx 1 root root 12616 Oct"..., 1024, 0
   x55982a6156b0) = 0x7ffd58369f80
strstr("-rwxrwxrwx 1 root root 12616 Oct"..., "rws") = nil
putchar(69, 2, 0, 0xd59ea72f)             = 69
...
putchar(110, 179, 0, 0xd59ea72f)          = 110
putchar(10, 110, 0, 0xd59ea72fError de permisos de ejecución
)           = 10
exit(-1 [no return ...]
+++ exited (status 255) +++
```

Con la nueva información descubierta se puede ejecutar en el terminal el comando ls obtenido y en su salida no se encuentra la cadena “rws”. Se puede ejecutar `chmod 4777 legado_original` para activar el sticky bit (el usuario puede ejecutar el archivo como root si es el dueño):

```
$ ls -l legado_original
-rwxrwxrwx 1 root root 12616 Oct 27 01:37 legado_original

$ chmod 4777 legado_original

$ ls -l legado_original
-rwsrwxrwx 1 root root 12616 Oct 27 01:37 legado_original

$ ./legado_original
Stocks v 3.03
Error acceso llave de protección
```

Al ejecutar el programa con los nuevos permisos se muestra un error diferente. Por lo que se ha vuelto a ejecutar con `ltrace`:

```
$ ltrace ./legado_original
...
[... fgets resumed] "-rwsrwxrwx 1 root root 12616 Oct"..., 1024, 0
   x55ac35ef66b0) = 0x7ffd148e2740
strstr("-rwsrwxrwx 1 root root 12616 Oct"..., "rws") = "rwsrwxrwx ...
pclose(0x55ac35ef66b0)                    = 0
ioperm(888, 2, 1, 2)                      = -1
...
putchar(10, 110, 0, -120Error acceso llave de protección
)                 = 10
exit(1 [no return ...]
+++ exited (status 1) +++
```

Esta vez avanza más en la ejecución, `strstr` ya no devuelve nil, se cierra el stream con `pclose` y `ioperm` intenta obtener permiso para usar un puerto, probablemente el puerto paralelo de la llave, pero falla mostrando otro error distinto. Al no disponer de un ordenador con drivers para un puerto paralelo no se puede avanzar mas en la ejecución. # 5. Paso 3: Exploración del código

Para comprender como funciona el programa y las comprobaciones de la clave hardware se ha desensamblado el código. Se han probado las diferentes herramientas listadas en la sección de [[#herramientas utilizadas]] y los problemas encontrados con ellas en [[#problemas encontrados]]. En esta sección se explica el uso de el desensamblador ghidra para explorar el código y en la siguiente sección se modificará el programa con dicha herramienta.

5.1 Ghidra

Ghidra es una suite de herramientas para ingeniería inversa de código libre desarrollada por la NSA. Soporta multiples arquitecturas y plataformas. En esta practica se ha utilizado para un ejecutable para x64 y Linux.

Al abrir ghidra, crear un proyecto, añadir el archivo ejecutable y hacer doble click sobre el se muestra una ventana ofreciendo la opción de analizar el archivo. Una vez analizado (con las opciones por defecto), se abre la ventana principal, mostrando el código desensamblado y el decompilado en pseudo C.

5.2 Función 1: main

En el listado de funciones se puede navegar fácilmente entre ellas y empezar a ponerles nombres. La función del siguiente ejemplo se puede interpretar que es la función main, además de que contiene la cadena “Stocks v 3.03” la cuál se mostraba por pantalla al ejecutar el archivo. Ghidra permite renombrar funciones y variables para hacer más sencillo el análisis.

```
1  // Fragmento original
2  undefined8 FUN_001009e0(undefined8 param_1,undefined8 *param_2)
3  {
4    ...
5    local_40 = *(long *)(in_FS_OFFSET + 0x28);
6    puts("Stocks v 3.03");
7    FUN_00101480(*param_2);
8    ...
9  }
10
11 // Fragmento renombrado
12 int main(int argc,char **argv)
13 {
14   ...
```

```
15    local_40 = *(long *)(in_FS_OFFSET + 0x28);
16    puts("Stocks v 3.03");
17    FUN_00101480(*argv);
18    ...
19  }
```

5.3 Llamadas al sistema y libC

Antes de renombrar las demás funciones, se puede observar el uso de diferentes llamadas al sistema y libc:

- puts: escribe una string a stdout.
- ioperm: establece permisos de e/s de un puerto.
- in/out: e/s en un puerto de bajo nivel.
- exit: termina el proceso con un código .
- system: ejecuta un comando de consola.
- putchar: escribe un byte a stdout.
- popen/pclose: abre/cierra un proceso creando una pipe, hace fork e invoca la consola con su primer parámetro.
- __sprintf_chk: formateo de una string.
- strstr: busca una string dentro de otra.
- scanf: lee de stdin con formato.

Gracias al enunciado del problema se sabe que la comprobación se realiza con un puerto paralelo por lo que el uso de `ioperm`, `in` y `out` podría ser para ese fin. `ioperm` se llama con tres parámetros: `ioperm(0x378,2,1)` siendo el primero de ellos el puerto. Buscando en internet el número del puerto se ha descubierto que, en efecto, el puerto pertenece históricamente al puerto paralelo LPT1 o LPT2.

5.4 Función 2: Comprobación de permisos

Continuando con las funciones, la siguiente recibe como parámetro `*argv` o lo que es lo mismo el nombre del fichero ejecutado. En un vistazo rápido salta a la vista la similitud con las llamadas obtenidas con `ltrace` en la sección [[#paso 2 Ejecución del programa]].

Empezando por `__sprintf_chk(acStack_428, 1, 0x400, "ls -l %s", file_name );`, el resultado de este método es un comando para obtener información sobre el archivo ejecutado. La siguiente linea crea un stream con la salida de la ejecución del comando. Mas adelante en la función se lee de ese stream y comprueba si contiene la string “rws” mediante `strstr(ls_string_buffer, "rws");`, si no contiene la string, imprime el mensaje de error de permisos obtenido anteriormente

(ver apartado [[#impresiones ofuscadas]]) y termina el programa con un `exit(-1);`. La función queda así:

```
// Comprobaciones de stack omitidas.
void comprobar_sticky_bits(char *file_name)
{
  FILE *__stream;
  char *rws_found_ptr_or_nullptr;
  char ls_string_and_result_buffer [1032];

  local_20 = *(long *)(in_FS_OFFSET + 0x28);
  __sprintf_chk(ls_string_and_result_buffer,1,0x400,"ls -l %s",
      file_name);
  __stream = popen(ls_string_and_result_buffer,"r");
  if (__stream == (FILE *)0x0) {  // Exit si popen falla
    FUN_00101560(&DAT_003023c0,3); // Print
    exit(-1);
  }
  fgets(ls_string_and_result_buffer,0x400,__stream);
  rws_found_ptr_or_nullptr = strstr(ls_string_and_result_buffer,"rws");
  if (rws_found_ptr_or_nullptr != (char *)0x0) { // Exit si strstr
      falla
    pclose(__stream);
    return;
  }
  FUN_00101560(&DAT_003028e0,2); // Print
  exit(-1);
}
```

5.5 Impresiones ofuscadas

El código oculta las cadenas que se muestran mediante ofuscación sencilla. Existen dos “estilos” de ofuscación, el primero trata de dividir los números de un array entre dos o tres y mostrar el resultado con un `putchar` como se puede ver en la siguiente función:

```
void print_divided_ints(int *message_array,int divisor)
{
  int current_char;
  int *next_char_ptr;

  current_char = *message_array;
  if (current_char != 0) {
    next_char_ptr = message_array + 1;
    do {
      putchar((int)(char)(current_char / divisor));
      current_char = *next_char_ptr;
      next_char_ptr = next_char_ptr + 1;
    } while (current_char != 0);
```

```
14      return;
15    }
16    return;
17  }
```

El segundo utiliza bit shifts y multiplicaciones como método de ofuscación. Un ejemplo sería `putchar((int)(char)((char)((ulong)((long)current_char * 0x55555556)>> 0x20)- (char)(current_char >> 0x1f)));`.

Al ser una encriptación muy simple, se puede obtener el mensaje original con relativa facilidad. El proceso ha sido: 1. Navegar a la dirección de memoria del array. 2. Seleccionar todas las direcciones bajo su etiqueta. 3. Cambiar la vista a int (4 bytes). 4. Crear y utilizar un script de python para desencriptar la memoria de la manera apropiada. 5. Renombrar la variable del array.

5.6 Cadena “Opción:”

El ultimo apartado de este paso consiste en entender la funcionalidad del programa más que el cómo evitar las comprobaciones.

Anteriormente, mediante el uso del comando `strings` en utf-8 (por las tildes) se encontró la cadena “Opción:” la cuál no aparece en ghidra. Buscando por la sección de memoria en la que se encuentran las demás cadenas encontradas, aparece una serie de bytes que en ascii escriben `Opci n:`. Utilizando la funcionalidad de referencias de ghidra, es posible navegar a los usos de la cadena, en este caso solamente se utiliza en un lugar:

```
1  // main, fragmento original
2  ...
3  __printf_chk(1,&DAT_00101645);
4  scanf("%d",&local_5c);
5  if (DAT_0030297c == 0) { // Irrelevante en este ejemplo
6      ...
7  else {
8      if (local_5c == 1) {
9          ...
10
11 // main, fragmento renombrado
12 ...
13 __printf_chk(1,s_opcion);
14 scanf("%d",&numero_opcion);
15 if (DAT_0030297c == 0) { // Irrelevante en este ejemplo
16     ...
17 else {
18     if (numero_opcion == 1) {
19         ...
```

Este fragmento del código separa la parte de mostrar el menu de opciones con la parte de cada opción.

Todas las opciones tienen una estructura similar, limpian la pantalla mediante `system("clear");` muestran algo por pantalla y esperan la entrada de dos caracteres para volver al menu de opciones (excepto la opción 7, que hace un **`return`** `0;`, terminando satisfactoriamente el programa).

6. Paso 4: Parcheo del programa

Una vez comprendido el programa se puede empezar a modificar para evitar el uso de la llave hardware. Como se ha descubierto anteriormente, las comprobaciones se hacen mediante `out` e `in` que sería lo equivalente a escribir en el puerto paralelo y leer su contestación, el problema es que no se dispone de esa clave a si que se puede intentar eliminar todas las llamadas al puerto y evitar el salto al código de error.

Para esto se ha utilizado la herramienta de parchear instrucciones de ghidra en el caso de los saltos condicionales y un script para ghidra el cuál hace más sencillo la transformación de instrucciones en NOPs, eliminando su comportamiento. Por ejemplo:

```
 1  // C original
 2  out(0x378,0x4d);
 3  in_value = in(0x378);
 4  if (in_value != -0x2b) {
 5
 6  // C objetivo
 7  if (in_value == -0x2b) {
 8
 9  // asm original
10  00100a5f 89 c8           MOV        in_value,ECX
11  00100a61 ee              OUT        DX,in_value
12  00100a62 ec              IN         in_value,DX
13  00100a63 3c d5           CMP        in_value,0xd5
14  00100a65 0f 85 1e        JNZ        LAB_00100f89
15           05 00 00
16
17  // asm modificado
18  00100a5f 89 c8           MOV        in_value,ECX
19  00100a61 90              NOP
20  00100a62 90              NOP
21  00100a63 90              NOP
22  00100a64 90              NOP
23  00100a65 0f 84 1e        JZ         LAB_00100f89
24           05 00 00
25
26  // C obtenido
27  if (in_value == -0x2b) {
```

Una vez eliminados todos los out/in e invertido sus saltos se puede eliminar también la llamada a

`ioperm` (sin haberla llamado, los out/in generan segmentation fault) y a la función de comprobación de permisos.

Para eliminar una llamada a una función también se puede sobrescribir su primera instrucción por RET y para hacer que un condicional siempre salte, colocar una comparación determinista delante del salto.

Una vez realizados estos cambios a lo largo de todo el programa se ha conseguido exportar un ejecutable modificado, el cual no necesita ni permisos ni llave hardware. Al ejecutarlo funciona perfectamente:

```
$ ./legado_modificado

<clear>
----STOCKS--3.03-------------

1) Crear nuevo fichero de stocks

2) Nueva entrada

3) Modificar entrada

4) Eliminar entrada

5) Listado entradas

6) Guardar entradas

7) Salir

Opción: 3

<clear>
Modificar entrada

-- Pulsa ENTER --
```

7. Problemas encontrados

Algunos de los problemas encontrados han sido los siguientes:

- El desconocimiento de los desensambladores, tanto el como utilizarlos como el sobrestimar sus capacidades. Esto ha causado frustración y ha llevado tiempo probar las diferentes herramientas para comprobar sus funcionalidades y decidir cuál utilizar principalmente.
- No se ha conseguido depurar instrucción por instrucción mediante un desensamblador. Se ha

intentado mediante gdb, lldb y/o frida en las diferentes herramientas tanto en local como en remoto, desde linux nativo y WSL sin éxito.
- Ninguno de los desensambladores permitía decompilar a código en C compilable, IDA Free lo permitía función a función, pero la calidad del código generado era muy baja. Leyendo algunas opiniones, IDA Pro es la única herramienta que exporta el programa completo a lenguaje compilable, pero es de pago.
- Las instrucciones in/out generan segmentation fault cuando no tienen permiso para escribir por lo que se han tenido que eliminar todas sus ocurrencias.

8. Conclusiones

Desensamblar y modificar código maquina es muy costoso y requiere gran habilidad con varias herramientas para hacerlo efectivamente. Si el programa hubiese sido mas largo o estado más ofuscado y protegido habría sido exponencialmente mas difícil y se habría tenido que profundizar en las herramientas, sobre todo en scripting para, por ejemplo en este caso, transformar todos los in/out en NOPs e invertir la condición siguiente automáticamente.

Desensamblar y modificar bytecode, ya sea de java o C#, parece más sencillo de realizar al ser un nivel mas alto y no dependiente de una plataforma o arquitectura especifica.

En un caso real, es probable que sea mas rápido y barato volver a desarrollar o comprar el programa que modificar el ya existente. Por no hablar de los riesgos de seguridad y estabilidad que pueden ser introducidos.

I. Referencias

[1] Parallel port - Wikipedia: https://en.wikipedia.org/wiki/Parallel_port#Interfaces