Práctica 4.- Bomba Digital - desensambladores

1 Resumen de objetivos

Con esta práctica se pretende profundizar en el uso de depuradores a bajo nivel y habituarse al uso de desensambladores y editores hexadecimal. Al finalizar esta práctica, se debería ser capaz de:

- Leer e interpretar lo que hace un programa sencillo sin información de depuración consultando y ejecutando su código binario.
- Modificar un programa del que sólo se dispone de su código binario sin información de depuración.
- Usar desensambladores y editores hexadecimal.

2 Herramientas

Se trabajará en Linux utilizando como compilador `gcc`. Para desensamblar se pueden usar desensambladores como `objdump` [2] [3] o depuradores como por ejemplo, `gdb` [4] [5], `ddd` [6] [7], o el entorno Eclipse. Los depuradores generalmente incorporan un desensamblador.

Aunque los depuradores son generalmente más útiles si disponen de la información de depuración generada con `gcc -g`, también pueden ejecutar paso a paso un ejecutable arbitrario, incluso cualquier secuencia de instrucciones a las que pueda saltar el contador de programa. Para los objetivos de esta práctica nos concentraremos en ejecutables que hayan sido compilados con la opción **`-no-pie`** y sin haber descartado los símbolos usando ***strip***. Aunque es posible depurar programas **`-pie`** y/o ***stripped*** [8] [9], preferimos concentrarnos en habilidades ejercitadas por la asignatura EC (relación C–ASM) y no tanto en las de otras asignaturas (carga y ejecución de programas, ASLR, etc.). Según el depurador preferido, lanzar el programa requerirá diferentes preparativos.

Con gdb se puede usar `disas main` o, aún mejor, cambiar a `layout asm` y `layout regs` para visualizar permanentemente en la ventana superior el desensamblado y los registros, y proceder a depurar con normalidad. Se pueden insertar *breakpoints* con especificaciones de línea (`br main`, `br boom`) o con direcciones (`br *main+64`, `br *0x40079b`). Seguramente interesará seguir las recomendaciones del manual sobre *scripts de inicialización* [10] para crear un fichero `~/.gdbinit` conteniendo `add-auto-load-safe-path ~/estruct/p4/bomba-gdb.gdb`, y dentro de este último fichero incorporar las instrucciones `layout asm`, `layout regs`, `br main`, `run`, y cualesquiera otras que deseemos ejecutar automáticamente al inicio de la sesión (ver Figura 5). Recordar que en lenguaje máquina usamos `nexti`/`stepi` en lugar de `next`/`step`, y que se puede cambiar el foco a la consola gdb pulsando **<Ctrl>-x o**.

Con ddd se puede ejecutar paso a paso un programa aunque no se haya compilado con información de depuración usando la ventana de código máquina *View→ Machine Code Window*. Si no aparece el código máquina en la ventana de código máquina, comprobar de nuevo si está activada *View→ Machine Code Window* (es frecuente confundirse con *View→ Source Code Window*) y/o probar a teclear en la consola gdb los comandos `info line main` ó `disas main`. De esta forma aparecerá en la ventana de código máquina y/o en la consola el código de la función que se indique. Para visualizar el código en un rango de direcciones, en la consola gdb se puede escribir `disas` *`start`*`,` *`end`* ó `disas` *`start`*`,+`*`length`*. Recordar que el paquete ddd sigue bloqueado en la versión 3.3.12 en Ubuntu.

Con Eclipse-CDT-9.4.3 (para Eclipse *Oxygen*) se puede importar un ejecutable sin información de depuración usando el asistente *File→ Import→ C/C++ Executable*, escogiendo *GNU Elf Parser* y el ejecutable *bomba* deseado. Tal vez deseemos usar antes *Window→ Perspective→ Close all Perspectives* para abandonar la sesión de trabajo anterior, e incluso *Window→ Perspective→ Open Perspective* para borrar proyectos anteriores usando el *Project Explorer*. Al confirmar pulsando el botón *Debug*, el editor nos muestra la posibilidad de visualizar el desensamblado (podemos reducir el tamaño de la ventana editor para agrandar la de desensamblado), y tenemos a nuestra disposición los botones normales (*Debug*, *Run*, *Terminate*, *Resume*, *Step Into*, *Step Over*...). Seguramente querremos activar el botón *Instruction Stepping Mode* para indicar a Eclipse que estamos depurando un desensamblado sin fuentes.

Prácticas de Estructura de Computadores, por Javier Fernández y Mancia Anguita, licencia CC BY-NC-ND.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

Recordar que el paquete Eclipse sigue bloqueado en la versión CDT-8.6 (*Luna*) en Ubuntu, y preferimos usar la versión 9.4.3 (*Oxygen/Photon*) disponible como aplicación *snap* en *Ubuntu Software*.

Una opción muy útil de los depuradores, cuando se busca algo en un código binario, es la que permite visualizar cualquier zona de memoria con diferentes formatos. Con gdb el comando `x` (*Examine*, consultar `help x`) sirve a tal propósito. Con ddd la opción equivalente es *Data→ Memory*, que visualiza gráficamente el resultado en la ventana de datos. Con Eclipse, la opción *Window→ Show View→ Registers/Memory* permite visualizar el valor de los registros del procesador y de las posiciones de memoria deseadas, en el formato (*rendering*) que se indique.

Para desensamblar se puede usar también `objdump` que está incluido en las *binutils* [3] (ejecutar `objdump --help` para ver sus opciones). El programa `readelf` de *binutils* da información sobre la cabecera ELF (*Executable Linux Format*) de los ficheros ejecutables. Otros programas también pueden ser útiles para obtener información relevante de un programa ejecutable.

Para poder modificar un programa del que sólo tenemos su código binario sin información de depuración se pueden usar editores de hexadecimal [11]. Hay diversos editores de hexadecimal gratuitos tanto para Linux (por ejemplo, *ghex*, *hexedit*) como para Windows (por ejemplo *WinHex*, *wxHexEditor*). En las aulas de prácticas se puede, por ejemplo, usar *ghex* (*GNOME Hex Editor*) ejecutando `ghex`. En el portátil con Ubuntu, se puede uno instalar *ghex*, o *hexedit*, o *ncurses-hexedit* (y ejecutar `ghex`, `hexedit`, o `hexeditor`). Otra opción es descargar e instalar *wxHexEditor* en Ubuntu o Windows [12].

3 Ejemplo tutorial

Para motivar el uso del desensamblador y del editor hexadecimal, proponemos el ejercicio de la *bomba digital*: un programa que solicita contraseña y pin, simulando una *explosión* si no se acierta, o se tarda demasiado tiempo en hacerlo. La bomba se le pasaría a otro compañero de clase, para ver si es capaz de *desactivarla* para que no le explote, o incluso *reactivarla* (modificarla) para que nos explote a nosotros.

Eso requerirá probablemente ejecutar el programa con el depurador, descubrir dónde y cómo se comprueban las contraseñas, y modificar el ejecutable (no disponiendo del fuente). La bomba se pasa en código ejecutable, naturalmente, compilada sin información de depuración, para dar menos pistas.

Compilar y ejecutar el programa mostrado en la Figura 1, deduciendo del código fuente cuál es la contraseña, el pin, y el máximo tiempo que se puede tardar en cada entrada de datos.

```
// gcc -Og bomba.c -o bomba -no-pie -fno-guess-branch-probability

#include <stdio.h>    // para printf(), fgets(), scanf()
#include <stdlib.h>   // para exit()
#include <string.h>   // para strncmp()
#include <sys/time.h> // para gettimeofday(), struct timeval

#define SIZE 100
#define TLIM 5

char password[]="abracadabra\n";     // contraseña
int  passcode  = 7777;               // pin

void boom(void){
       printf( "\n"
               "***************\n"
               "*** BOOM!!! ***\n"
               "***************\n"
               "\n");
       exit(-1);
}

void defused(void){
       printf( "\n"
               ".......................\n"
               "··· bomba desactivada ···\n"
               ".......................\n"
               "\n");
       exit(0);
}
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```c
int main(){
        char pw[SIZE];
        int  pc, n;

        struct timeval tv1,tv2;       // gettimeofday() secs-usecs
        gettimeofday(&tv1,NULL);

        do      printf("\nIntroduce la contraseña: ");
        while ( fgets(pw, SIZE, stdin) == NULL );
        if    ( strncmp(pw,password,sizeof(password)) )
            boom();

        gettimeofday(&tv2,NULL);
        if    ( tv2.tv_sec - tv1.tv_sec > TLIM )
            boom();

        do  {   printf("\nIntroduce el pin: ");
         if ((n=scanf("%i",&pc))==0)
                scanf("%*s"    )==1;            }
        while ( n!=1 );
        if    ( pc != passcode )
            boom();

        gettimeofday(&tv1,NULL);
        if    ( tv1.tv_sec - tv2.tv_sec > TLIM )
            boom();

        defused();
}
```

Figura 1: bomba.c: bomba digital con 2 fases: *strings* (contraseña) y enteros (pin)

Comprobar el funcionamiento de todas las ramas de control (sentencias `while/if`), probando a:

- Introducir EOF como contraseña pulsando <Ctrl>-D (1er `while`)
- Introducir una contraseña incorrecta, por ejemplo <Enter>, "hola" o "abracadabra 123" (1er `if`)
- tardar mucho en ese primer paso (siendo la contraseña *correcta*, 2º `if`)
- Introducir EOF como pin pulsando <Ctrl>-D, o pulsar <Enter>, o "a b c d7777" (3er `if`, 2º `while`)
- introducir un pin incorrecto, por ejemplo "123" (4º `if`)
- tardar mucho en introducir el pin *correcto* (5º `if`)
- Introducir el pin correcto, o variantes como por ejemplo "a b c 7777 d", o "7777abcd"

Aunque pasemos al compañero esta *bomba* como ejecutable sin información de depuración, él podría obtener diversa información (usando las distintas utilidades *binutils* o ejecutando paso a paso con el depurador) que le ayudara a estudiar el funcionamiento de nuestra bomba. Podría eliminar las comprobaciones de contraseñas, o incluso descubrir las contraseñas empleadas (si se ponen algunas condiciones mínimas como por ejemplo que no haya cálculos irreversibles de tipo criptográfico). Podríamos utilizar otras herramientas (***strip***) para eliminar del ejecutable aún más información, aunque en esta práctica no lo permitiremos para no dificultar demasiado la solución de la misma.

Conociendo las claves, el compañero puede evitar la explosión de la bomba. Utilizando un editor hexadecimal, puede modificar el ejecutable de forma que no sea necesario introducir claves, o aún más, cambiar las claves para que nos explote a nosotros.

Ejercicio 1: evitar las comprobaciones

Compilar el programa de la Figura 1 sin información de depuración (y sin **–pie**, con `gcc -Og bomba.c -o bomba -no-pie -fno-guess-branch-probability`), y ejecutarlo paso a paso con `gdb` para evitar las comprobaciones. Probablemente sigamos un proceso deductivo similar a éste:

- Poner un *breakpoint* en la primera instrucción (`br main`) y empezar la ejecución (`run`). Si no se siguió la recomendación de crear un fichero `~/.gdbinit` conteniendo el comando `add-auto-load-safe-path` y otro `bomba-gdb.gdb` conteniendo los cuatro comandos (ver Figura 5), será necesario cambiar a `layout asm` para poder ver las direcciones y el desensamblado, y conveniente añadir `layout regs` para visualizar permanentemente el valor de los registros.
- Ir avanzando con `nexti` (para no meternos en subrutinas). Notar que aunque compilamos sin información de depuración, podemos ver los nombres de variables y las funciones invocadas.
- En algún momento se nos pedirá la contraseña. Introducir algún valor que podamos reconocer después, como por ejemplo "`hola`".

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

- Seguir avanzando con `nexti`. La bomba explota, en este caso dentro de una función llamada `boom`. Poner un *breakpoint* en dicha llamada (`br *main+117`) para futuras ejecuciones.
- Volver a ejecutar (`run`), aprovechando la parada en el primer *breakpoint* para eliminarlo (`del 1`, no necesitamos pararnos más allí). Continuar con `cont` hasta volver a la llamada a `boom`.
- Nos metemos dentro con `stepi`, pero vemos que no hay nada que aprender: sencillamente se imprime "`BOOM!`" y se llama a `exit`, así que la decisión de explotar ya se había tomado. *El programa llama a boom cuando ya ha decidido que hay que explotar*.
- Volvemos a ejecutar y leemos, poco antes de la llamada a `boom`, que se invoca a `strncmp` y si el resultado es cero se evita la llamada a `boom`. En un ejercicio posterior consultaremos la página de manual de `strncmp` para entender la bomba, pero ahora nos basta con evitar que ésta explote, lo cual se puede conseguir fácilmente parando el programa en la instrucción `test` (`br *main+113`) y modificando entonces el valor de EAX (`set $eax=0`). Podemos dejar el *breakpoint* de `test` y eliminar el de `call boom` (`del 2`) para futuras ejecuciones.
- Pasado este peligro, seguimos avanzando con `nexti` hasta ver que vuelve a llamar a `boom`, esta vez dependiendo de alguna comprobación relacionada con `gettimeofday`. Luego consultaremos los manuales correspondientes. Igual que antes, podemos añadir un *breakpoint* en la instrucción `cmp $0x5,%eax`, para futuras ejecuciones. Seguramente desearemos ajustar `set $eax=0` antes de ejecutar `cmp`.
- Pasado este segundo problema, en algún momento se nos pedirá el pin. Introducir algún valor que podamos reconocer después, como por ejemplo "`123`".
- Con la experiencia adquirida, poco después comprobamos que hubiéramos debido introducir el `7777` que vemos en EAX, en lugar del `123` que ahora vemos en `x/1dw(0xc+$rsp)` o con `p*(int*)(0xc+$rsp)`. Añadimos un *breakpoint* en la instrucción `cmp %eax,0xc(%rsp)`, para poder modificar también en ejecuciones futuras `set $eax=123` (o introducir `7777` como código) y pasar esta comprobación sin problemas.
- Pasado este tercer peligro, un poco más adelante volvemos a ver `cmp $0x5,%eax`, tras una llamada a `gettimeofday` y antes de `call boom`. Ya sabemos cómo arreglar el problema. Dejamos también un *breakpoint* anotado por si tuviéramos que reiniciar el programa.
- Por último se invoca `defused` y comprobamos que ya no hay más bombas.

El proceso es entretenido y conviene anotar los diversos escollos para poder volver rápidamente al punto donde nos quedamos si por alguna equivocación nos explotara la bomba. A lo mejor este ejemplo es tan sencillo que puede memorizarse sin anotaciones, pero algunos casos pueden llegar a ser tan complejos que necesiten automatización mediante *scripts* `gdb`.

Ejercicio 2: ejecutable sin comprobaciones

Hemos localizado 4 posiciones en el ejecutable en donde se invoca a *boom*, con comprobaciones previas. En un ejercicio posterior estudiaremos el significado de las comprobaciones. Ahora nos interesa crear un ejecutable que no explote, sin necesidad de ejecutarlo paso a paso con gdb. Aunque hemos estudiado los cuatro saltos, modificaremos sólo los relacionados con las claves (la contraseña y el pin).

```
 ...
0x4007c0 <main+101>     lea    0x2008a1(%rip),%rsi       # 0x601068 <password>
0x4007c7 <main+108>     callq  0x4005d0 <strncmp@plt>
0x4007cc <main+113>     test   %eax,%eax
0x4007ce <main+115>     je     0x4007d5 <main+122>
0x4007d0 <main+117>     callq  0x400727 <boom>
0x4007d5 <main+122>     ...
 ...
0x4007df <main+132>     callq  0x4005f0 <gettimeofday@plt>
0x4007e4 <main+137>     mov    0x20(%rsp),%rax
0x4007e9 <main+142>     sub    0x10(%rsp),%rax
0x4007ee <main+147>     cmp    $0x5,%rax
0x4007f2 <main+151>     jle    0x4007f9 <main+158>
0x4007f4 <main+153>     callq  0x400727 <boom>
0x4007f9 <main+158>     ...
 ...
0x400820 <main+197>     callq  0x400620 <__isoc99_scanf@plt>
 ...
0x400841 <main+230>     mov    0x200819(%rip),%eax       # 0x601060 <passcode>
0x400847 <main+236>     cmp    %eax,0xc(%rsp)
0x40084b <main+240>     je     0x400852 <main+247>
0x40084d <main+242>     callq  0x400727 <boom>
0x400852 <main+247>     ...
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

```
   ...
   0x40085c <main+257>     callq  0x4005f0 <gettimeofday@plt>
   0x400861 <main+262>     mov    0x10(%rsp),%rax
   0x400866 <main+267>     sub    0x20(%rsp),%rax
   0x40086b <main+272>     cmp    $0x5,%rax
   0x40086f <main+276>     jle    0x400876 <main+283>
   0x400871 <main+278>     callq  0x400727 <boom>
   0x400876 <main+283>     callq  0x400741 <defused>
   0x40087b                nopl
   ...
```

Figura 2: ventana de código desensamblado mostrando las 4 invocaciones a boom()

Podemos consultar un desensamblado del programa en esas 4 posiciones, para comprobar que las instrucciones de salto se codifican con dos bytes: uno para el *codop* (`0x74` el de JE, `0x7e` el de JLE), y otro para la dirección de salto, que lleva direccionamiento relativo a contador de programa, indicando en los cuatro casos un salto hacia adelante de 5 posiciones, las que ocupa la llamada a *boom*.

```
...
4007ce:      74 05                 je     4007d5 <main+0x7a>
4007d0:      e8 52 ff ff ff        callq  400727 <boom>
...
4007f2:      7e 05                 jle    4007f9 <main+0x9e>
4007f4:      e8 2e ff ff ff        callq  400727 <boom>
...
40084b:      74 05                 je     400852 <main+0xf7>
40084d:      e8 d5 fe ff ff        callq  400727 <boom>
...
40086f:      7e 05                 jle    400876 <main+0x11b>
400871:      e8 b1 fe ff ff        callq  400727 <boom>
...
```

Figura 3: desensamblado de los saltos condicionales

En los manuales de Intel podemos comprobar que existe un salto *incondicional* con ese mismo modo de direccionamiento (*codop* `0xeb`), de manera que si pudiéramos sustituir en el fichero ejecutable esos 4 *codops* por `0xeb`, daría igual que las claves fueran erróneas, porque de todas formas se saltaría la explosión. De hecho, si tecleamos las claves rápido (0 y 0, por ejemplo), tardaremos menos de los 5 segundos permitidos, y ni siquiera necesitaríamos modificar los *codops* `jle`, sino tan solo los `je`.

Para realizar las modificaciones deberíamos teclear los comandos gdb mostrados en la Figura 4. Un problema es que al cargar `file bomba`, se lanzará automáticamente el fichero *script* `bomba-gdb.gdb` de la Figura 5. Para evitarlo, desearíamos renombrar temporalmente dicho script (a `bomba-gdb.gdb-`, por ejemplo). Una vez hecho eso, podríamos teclear los comandos de la Figura 4, o mejor salvarlos en un *script* `bomba.gdb` y lanzarlo desde gdb con el comando `source bomba.gdb`, o mejor aún, lanzar desde línea de comandos `gdb -q -x bomba.gdb`.

```
# P4,E2:ejecutable sin comprobaciones

# Automatizar modifs a los dos saltos
# Pensado para "source bomba.gdb"
# desde shell: gdb -q -x bomba.gdb
# renombrar fichero "bomba-gdb.gdb"
# que no se cargue auto "file bomba"

### permitir escribir en ejecutable
    set write on
### reabrir ejecutable con perm. r/w
    file bomba
### comprobar direcciones a cambiar
    x/i 0x4007ce
    x/i 0x40084b
### realizar los cambios
    set {char} 0x4007ce=0xeb
    set {char} 0x40084b=0xeb
### comprobar instrucciones cambiadas
    x/i 0x4007ce
    x/i 0x40084b
### salir para desbloquear ejecutable
    quit
```

Figura 4: script bomba.gdb, ejecutar con source

```
# Prac.4, Ej.1: evitar las comprobaciones

# Ahorrarse teclear esdto cada vez
# que se empieza a depurar la bomba

# Debido al nombre fichero "bomba-gdb.gdb"
# se carga automaticamente con "gdb bomba"

layout asm
layout regs
br main
run
```

Figura 5: bomba-gdb.gdb , renombrar temporalmente

Conviene probar las tres formas (copiar línea a línea, `source script`, y `gdb -q -x script`).

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Ejercicio 3: editor hexadecimal

Se puede ejecutar en un terminal la bomba modificada en el ejercicio anterior para comprobar que efectivamente no explota, incluso tecleando "*hola*" como contraseña y "*123*" como código (o más rápido de teclear, "*0*" y "*0*"). De todas formas, es mucho más cómodo utilizar un editor hex para modificar (incluso localizar) los códigos de operación. Recompilar la bomba (`gcc -Og bomba.c -o bomba -no-pie -fno-guess-branch-probability`), y ejecutar `ghex bomba` para entrar en el editor. En la ventana derecha se edita en ASCII, en la ventana izquierda en hexadecimal, y abajo se ven simultáneamente diversas conversiones de interés.

Se pueden aplicar diversos métodos para repetir la modificación de la bomba realizada anteriormente con gdb. Una posibilidad sería la siguiente (ver Figura 3):

- La secuencia de bytes a partir del primer JE que queremos modificar es `74 05 e8 52`. Con un poco de suerte, esa secuencia será única en el fichero ejecutable (opción *Edit→Find*). Una vez encontrada, basta con editar el primer `74`→`eb` (JE→JMP) en la ventana hex.
- Similarmente con la segunda secuencia `74 05 e8 d5`.

Otra posibilidad inmediata sería localizar el offset exacto de la instrucción en el fichero, conociendo el offset de la sección `.text` y la diferencia de posiciones entre `.text` y la instrucción a modificar. No hay límites (salvo la imaginación de cada uno) a los distintos métodos que se pueden usar para localizar y modificar el *byte* deseado con el editor hexadecimal.

Eso sí, para estar seguros de la modificación realizada, se debe comprobar (ejecutando desde línea de comandos Linux) que efectivamente la bomba no explota.

Ejercicio 4: descubrir las claves

En los ejercicios anteriores nos hemos limitado a modificar 2 de las 4 bifurcaciones condicionales porque sabíamos (por haber visto el código fuente) que las otras 2 sólo se activan si se gastan más de 5 segundos tecleando las claves. Ese conocimiento también se puede obtener consultando el desensamblado del ejecutable y las páginas de manual de las funciones invocadas. Así:

- `gettimeofday()` tiene un primer argumento puntero a `struct timeval`, y un segundo usualmente NULL. En nuestro caso, `&tv1` se traduce como `0x10(%rsp)`.
- El argumento de `printf()` es `0x4009c8`, *string* pidiendo la contraseña (comprobarlo con `p(char*)$rsi`). El interfaz de `__printf_chk` añade un primer argumento `int flag` indicando si se ha de comprobar desbordamiento de pila antes de calcular un resultado.
- `fgets()` copia en el *string* (1er arg.) hasta un máximo de n bytes (2º arg.) leidos del *stream* (3er arg.). En el desensamblado podemos comprobar que `n=0x64` y que la contraseña se va a almacenar en `0x30(%rsp)`. Tras la llamada, podemos imprimir `p(char*)(0x30+$rsp)`, o sencillamente `p(char*)$rax`.
- `strncmp()` compara alfabéticamente un primer argumento *string* (nuestro `0x30(%rsp)`) con otro segundo argumento *string* `0x601068` (que para mayor claridad aparece etiquetada como *password*) hasta una longitud máxima de `$0xd` indicada como tercer argumento (notar que el `sizeof(password)` indicado en el código fuente incluye el `\n` y `\0` finales). Si esa cadena y la que se leyó de teclado son iguales se salta la llamada a la bomba (`test/je/call`).
- Hay que tener muy poca curiosidad para no probar `p(char*)$rsi` antes de la llamada a `strncmp()`. También se puede probar `p(char*)0x601068` o `p(char[0xd])password`, o incluso `p{char[13]}0x601068`.
- Un estudio similar del código que sigue y la página de manual de `gettimeofday()` nos revela que haber tardado más de 5 segundos desde el inicio del programa causa llamada a la bomba.
- Más adelante se compara EAX, traido de `0x601060` (que para más claridad aparece etiquetado como *passcode*) con `0xc(%rsp)`, 2º argumento de `scanf()`. Hemos encontrado el pin. Para proseguir con el estudio podemos imprimir `p*(int*)(0xc+$rsp)` para recordar el entero que tecleamos, y poner ahora ese valor en EAX al objeto de poder saltar la bomba. También se puede imprimir `p*(int*)0x601060` o mejor `p(int)passcode` (o incluso `p{int}0x601060`) para recordar el pin correcto.
- El estudio del final del programa revela el mismo límite de 5 segundos para esta segunda parte. Notar que es la propia función `defused` la que se encarga de terminar el programa.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Como vemos, es muy conveniente disponer del nombre de las funciones llamadas desde el programa para, con la ayuda de los manuales correspondientes, comprobar los valores de los argumentos y entender las operaciones realizadas sobre ellos. Esto nos facilita enormemente comprender el funcionamiento de la bomba. Es posible eliminar dicha información del fichero ejecutable, aunque para los objetivos de esta práctica no recurriremos a dicha posibilidad.

Ejercicio 5: modificar las claves

Habiendo descubierto dónde están almacenadas las claves (y su valor), es inmediato modificarlas (sin alterar el tamaño del ejecutable, naturalmente), con el propio depurador o con el editor hexadecimal.

La opción *Edit→ Find* de ghex sería suficiente si la contraseña (buscando *abracadabra* en ASCII) y/o el pin (buscando `61 1e 00 00` en *little-endian*. *¿Por qué?*) sólo aparecen una vez en todo el ejecutable.

Si aparecen varias veces, tal vez convendría modificar el ejecutable desde gdb con `set*(char*) 0x601068='O'` o con `set*(int*)0x601060=123`. A lo mejor es preferible la sintaxis `set {char[13]}0x601068="hola,adios.\n"` o `set{int}0x601060=123`. También se puede usar `p (char[13])password` o `p(int)passcode` para comprobar si el cambio se ha realizado correctamente. Recordar (re)cargar el fichero tras haber activado `set write on`, como vimos en la Figura 4.

Observar que al haber indicado `sizeof(password)` como tercer argumento de `strncmp()`, el número de caracteres comparados es siempre 13: los 11 de *abracadabra* más el avance de línea (0xA) y el final de string (0x0). Si se hubiera indicado `strlen(password)`, el tamaño se recalcularía cada vez que se ejecutara el programa, lo cual permitiría cambiar la contraseña por otra más corta.

Aunque ghex lo permita (pulsando las teclas <Insert> o <Supr>/<Del>), en general no se debe incrementar ni decrementar el tamaño del programa porque afectaría al inicio de las secciones posteriores en el fichero, a las direcciones no independientes de la posición, e incluso a las direcciones de datos expresadas de forma relativa a contador de programa, si cambia la distancia entre ambos (dato y contador de programa) debido a una inserción o borrado intermedios.

En el caso particular de nuestra bomba, observamos mediante ghex que la variable `password` va seguida de la conocida sección `.comment` que contiene el texto de la directiva `.ident "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"`. Esto nos permitiría aumentar la longitud de `password` sobrescribiendo `.ident`, pero como en nuestro código fuente seguimos comprobando `sizeof(password)`, cualquier string que coincidiera en sus primeras 13 letras con el modificado mediante ghex evitaría la explosión, y según nuestras reglas eso no se consideraría una bomba válida, ya que las claves deben ser valores **fijos y únicos**. También se puede comprobar mediante `readelf –p .comment bomba` cómo afecta al comentario de identificación la sobrescritura realizada.

Una vez salvado el ejecutable modificado, la bomba está lista para devolverla a su autor.

4 Trabajo a realizar

4.1 Programar la bomba digital

Se trata de implementar un programa en C que simule la explosión de una bomba si el usuario no introduce correctamente la contraseña y el pin, o si los introduce en un tiempo superior a 1 minuto. No es necesario que la bomba explote mientras el usuario esté tecleando las claves; basta con que lo haga después de introducir cada clave, si es entonces cuando se detecta que ha tardado demasiado tiempo.

Las claves serán una contraseña (cadena) de entre 8 y 10 caracteres, y un pin (numérico) de entre 3 y 6 dígitos decimales. Para que se pueda pedir al usuario la segunda clave, éste debe haber acertado ya la primera en el tiempo estipulado. Si se desea, se puede comprobar el tiempo antes de comprobar la clave. No es necesario indicar si la explosión se debe a error en la clave o a expiración del plazo.

El **objetivo** no es que el código fuente sea difícil de entender, sino que el **desensamblado** sea **entretenido de resolver**, lo suficiente como para que los compañeros tarden **entre 15 minutos y una hora** en descubrir las claves. El texto se resalta para indicar que esta condición es muy importante.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

El programa se compilará sin información de depuración (sin `-g`), sin producir ejecutable independiente de la posición (con `-no-pie`) y sin librerías (salvo libC, incluida implícitamente), y sin aplicarle opciones o utilidades que eliminen más símbolos (tipo ***strip***). Se permitirá utilizar optimización hasta nivel 2, de manera que la orden de compilación más compleja será `gcc –O2 bomba.c -o bomba –no-pie`. Si se desea usar alguna otra opción de compilación, consultar con el profesor si es válido su uso.

No se aplicarán algoritmos con cálculos irreversibles de tipo criptográfico. **Las claves deben ser valores fijos, constantes** que no dependan de la fecha, la hora del día, el tiempo que se tarde en teclearlas, etc. También deben ser **valores únicos**, de forma que no debe haber varias contraseñas que eviten la explosión, ni tampoco varios pines, sino tan sólo uno correcto. Por supuesto que no deben ser valores generados aleatoriamente. Respecto a los algoritmos válidos, como norma general se puede usar cualquier técnica estudiada en clase. Los profesores serán los árbitros definitivos sobre qué ideas valen y cuáles no.

Sería válido, por ejemplo, usar la cifra del César para codificar la contraseña, o usar indexación en un *array* para transformar los dígitos del pin. Quedaría a juicio de los profesores decidir por ejemplo si es válido usar la sucesión de Fibonacci para codificar el pin, según qué valor(es) concreto(s) de la serie se desee(n) usar (dependiendo de si son o no lo suficientemente fáciles como para descubrir la clave en un tiempo razonable).

Para puntuar en esta fase y poder participar en la siguiente fase de resolución y modificación de bombas de compañeros, se debe **entregar** en *SWAD→ Evaluacion→ Actividades*, en la fecha que se indique:

1. El **ejecutable** generado (sin información de depuración, sin eliminar símbolos con utilidades)
2. El **fuente** C, indicando en un comentario en la primera línea la **orden de compilación usada**
3. **Explicación** del método preferido para desactivar/reactivar la bomba (fichero texto, fichero pdf, o *script gdb*). La explicación debe comenzar **revelando la contraseña y el pin**, originales y modificados (basta con cambiar una letra de la contraseña y un dígito del pin) y debería ser comprensible para cualquier compañero de clase que domine la materia explicada en la asignatura. En principio las explicaciones las verán sólo los profesores, no los compañeros.

El objetivo de la **explicación** no es contar cómo se ha redactado el código fuente, sino demostrar que era viable **descubrir y cambiar las claves** en el tiempo estipulado (**entre 15 minutos y una hora**). Eventualmente, si algún compañero reclama en la siguiente fase que una bomba no se puede resolver, se le mostrará la explicación correspondiente y se anulará la bomba si la explicación no resuelve las dudas sobre cómo se podían descubrir las claves. Se recomienda dar una explicación paso a paso que en una cantidad mínima de tiempo permita al lector obtener la solución, o aún mejor un *script gdb* profusamente comentado, que al ejecutarlo imprima por pantalla las claves originales y las modificadas.

Un posible **formato** recomendado para la explicación consistiría en indicar, para cada decisión tomada durante la sesión de depuración, la siguiente información:

a. Volcado **ensamblador** de dónde estamos parados (*copy-paste* del desensamblado)…
   i. mostrando dirección del *breakpoint*, o…
   ii. explicando cómo y por qué se llegó a la instrucción actual
b. Volcado *print* o *examine* o *registers* de la **información** relevante
c. **Razonamiento** de qué se hace a continuación (por qué se toma qué decisión)

Otra posibilidad más taxativa consistiría en diseñar un *script gdb* (al estilo del *explain.gdb* mostrado en la Figura 6) conteniendo las instrucciones requeridas para imprimir la contraseña y el pin por pantalla, razonando todo el proceso. Se esperaría que cada instrucción *gdb* llevara un comentario explicando la utilidad de la misma, de manera que leyendo los comentarios se pudiera seguir el proceso lógico seguido para descubrir las claves. Se esperaría que ejecutando `gbd –q -x explain.gdb` resultaran imprimidas por pantalla las claves originales y las modificadas. En el Apéndice 1 se muestra un posible *script* `explain.gdb` del proceso que hemos explicado en los ejercicios del Tutorial para descubrir y cambiar las claves de la bomba de ejemplo.

El **ejecutable** (sin información de depuración, sin `-pie` y en un .zip) también se depositará (además de en *Evaluacion→ Actividades*) en **Archivos Compartidos** del grupo de prácticas (*Asignatura→ Archivos Compartidos→ seleccionar grupo*) al inicio de la fase correspondiente, para que quede accesible por parte de los compañeros del grupo. Notar que cuando se seleccionan los Archivos Compartidos del grupo, dejan de verse los Archivos Compartidos de la asignatura, y viceversa.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

4.2 Resolver y modificar bombas digitales de los compañeros

Se trata de descubrir y cambiar las claves de un par de bombas digitales realizadas por otros compañeros (que también la hayan entregado en la fecha establecida). Para ello sólo se dispondrá del **ejecutable** del programa, que estará depositado en **Archivos Compartidos** del grupo de prácticas. Se deberán localizar las claves, o al menos indicar su valor, si por algún motivo no fuera viable indicar su dirección también. Como ayuda se pueden utilizar depuradores y desensambladores. Recordar que hay caracteres codificados con más de un byte, consultar la tabla de códigos UTF-8 en la Wikipedia (o buscarla con Google).

Cada estudiante desactivará dos bombas digitales (distintas de la suya), durante las dos horas de la sesión de prácticas correspondiente, pudiendo dejar para más tarde la redacción del fichero de explicaciones. De nuevo, la explicación más taxativa podría consistir en un *script gdb* que imprima en pantalla las claves originales y modificadas, siempre que incorpore comentarios que expliquen el proceso lógico seguido.

Para puntuar en esta fase, se debe **entregar** en *SWAD*→ *Evaluacion*→ *Actividades*, en la fecha que se indique, por cada bomba resuelta y/o modificada:

1. Ejecutable **original**
2. Ejecutable **modificado**
3. **Explicación** del método seguido. El fichero debe comenzar **identificando** la bomba asignada por el profesor (de qué compañero es), **revelando** la contraseña y el pin originales, y **revelando** la contraseña y el pin modificados. Posteriormente debe explicarse del **método** seguido para localizar o **descubrir** las claves, y el método seguido para **modificar** las claves. Estas explicaciones deben permitir al profesor deducir que el trabajo lo ha realizado uno mismo

Se recomienda usar para estas explicaciones el mismo formato usado para el apartado anterior en la explicación de la propia bomba, o incluso el mismo método de usar un *script gdb* razonando por qué se usa cada instrucción. Naturalmente, no es válido (puntúa 0) indicar solamente la solución (claves) sin aclarar cómo se ha obtenido.

Las bombas modificadas, al igual que las originales, sólo deben aceptar la contraseña fija y única y el pin fijo y único indicados en las respectivas explicaciones. Si alguna bomba original asignada por el profesor resultara admitir varias claves, el estudiante afectado debe avisar al profesor para que se le asigne otra bomba y se penalice al autor de la bomba original. El profesor procurará compensar al estudiante afectado por la pérdida de tiempo que le hubiera causado dicha bomba.

Las bombas deberían poderse resolver en una cantidad de tiempo variable entre 15min y 1hora. Cualquier estudiante que haya dedicado al menos 30min al estudio de una bomba y perciba que es demasiado difícil, tiene derecho a solicitar que se le asigne una bomba distinta con la que seguir trabajando (al objeto de cumplir con su cupo de 2 bombas), y a que posteriormente el profesor le envíe una copia del fichero de explicación del autor de la bomba que causó la dificultad. Si no quedara satisfecho con la explicación o con la dificultad de la bomba (debería poder resolverse en unos 15-60min), el estudiante avisará al profesor para que penalice al autor de la bomba original, y procure compensar al estudiante afectado por la pérdida de tiempo que le causó dicha bomba.

4.3 Resolver y modificar bombas digitales del profesor

En promociones anteriores se han experimentado diversos problemas con la realización de esta práctica. La mayoría de ellos desaparecen si los profesores disponen de una batería de bombas de dificultades crecientes, empezando por dificultad 5 (definida como “idéntica a la estudiada en clase”), dificultad 6 (“se resuelve igual de fácilmente pero el código no es idéntico al de clase”), dificultad 7 (“algo más difícil que la de clase”), etc. Algunos de los problemas a que nos referimos son, por ejemplo:

1. Algunos estudiantes no siguen las instrucciones. No comprueban que las claves de su bomba se puedan descubrir y cambiar en unos 15-60min. Entregan una explicación sobre cómo hicieron el código fuente, creyendo meritorio que éste sea difícil de leer, o que la bomba pida varias contraseñas o muchos pines. También se han presentado bombas que aceptan soluciones no únicas como por ejemplo cualquier pin que cumpla una cierta condición numérica fácil de satisfacer. Todas esas bombas causan retrasos importantes a los estudiantes a los que desafortunadamente les tocan en suerte.

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

2. Algunos estudiantes resuelven muchas bombas más rápido de lo razonablemente creíble, especialmente conforme se acerca el fin de la sesión. En casos extremos, grupos enteros de 25 estudiantes han dejado pasar 100 minutos de sesión sin actividad apreciable para pasar a resolver más de 2 bombas cada uno en los últimos 20 minutos. Estos patrones de actividad obligan a los profesores a tomar medidas defensivas como por ejemplo dejar de aceptar soluciones 10min antes del final de la sesión.
3. Siempre hay distintas opiniones sobre la dificultad de las bombas: algunos estudiantes opinan que una bomba es igual de fácil que la estudiada en clase, y otros que es mucho más difícil.
4. El mecanismo de seguridad de "revelar la explicación al estudiante afectado" ha probado ser inútil. En los contados casos en los que se ha invocado el derecho a consultar la explicación, el estudiante afectado ha renunciado a los pocos minutos a pedir compensación por el tiempo perdido, aceptando por tanto que la explicación fuera buena. Las dos explicaciones que se nos ocurren a los profesores para explicar estas situaciones son alarmantes.

Si los profesores disponen de una batería de 30 bombas de cada dificultad (con la definición esbozada anteriormente para las dificultades 5, 6, 7, etc…), al principio de la sesión se puede asignar una bomba diferente de dificultad 5 a cada estudiante, que debería resolverse en unos 10min (si el estudiante ha asistido con provecho al Tutorial de esta práctica) o tal vez unos 25min (si se entretiene en ir desarrollando simultáneamente el *fichero de automatización gdb*). Una vez comprobado que la bomba original no explota con las claves originales y sí con las modificadas, y que la bomba modificada no explota con las claves modificadas y sí con las originales (que deben diferenciarse en al menos un carácter y en al menos un dígito), se procede a asignarle al estudiante otra bomba de dificultad 6, y luego de dificultad 7, etc. El estudiante se limita a anotarse las soluciones y guardarse los ejecutables (preferiblemente en *SWAD→ Evaluacion→ Actividades*), para posteriormente de forma no presencial proceder a redactar los ficheros de explicación que también deben entregarse en SWAD, y dedicar todo el tiempo presencial a intentar llegar lo más lejos posible en las dificultades de las bombas. Algunos estudiantes preferirán incluso desarrollar sobre la marcha el *script gdb* anotando los comandos que van ejecutando y añadiendo los comentarios oportunos mientras que van resolviendo la bomba, de forma que no tengan que dedicarle tiempo posterior (no presencial) a redactar explicaciones adicionales.

Si todos los estudiantes entendieron el Tutorial, y los profesores se ajustan a la definición de "dificultad 5" y "dificultad 6", todos los estudiantes deberían estar solicitando bomba de dificultad 7 unos 20-50min después del inicio de la sesión, cumpliendo por tanto de sobra los objetivos de la Actividad.

5 Entrega del trabajo desarrollado – Posible etapa de competición entre grupos

Los distintos profesores de teoría y prácticas acordarán las normas de entrega para cada grupo, incluyendo qué se ha de entregar, cómo, dónde y cuándo. Esta práctica consiste en diversas fases de una única sesión (4.1, 4.2, 4.3), de forma que sólo quien asista a alguna sesión podrá entregar la Actividad correspondiente a dicha sesión.

Por ejemplo, puede que en un grupo el profesor indique que las explicaciones sólo pueden ser *scripts gdb* profusamente comentados, y que la Actividad 4.2 sólo sirve de entrenamiento, no para puntuar. Para corregir la Actividad 4.1 el profesor copia y pega la primera línea comentada en el fuente para producir el ejecutable, y posteriormente ejecuta gdb `explain.gdb`, comprobando que las claves originales y modificadas imprimidas funcionan como se espera. Para corregir la Actividad 4.3, para cada bomba resuelta el profesor ejecuta el script proporcionado como explicación para obtener las claves originales y modificadas, comprobando de nuevo que la bomba original acepta las primeras y explota con las segundas, mientras que con la bomba modificada (por haber ejecutado el script) sucede al revés.

Puede que en otro grupo el profesor de prácticas proporcione otras instrucciones distintas.

En cualquier caso, **el examen de prácticas** a final del curso (Test TP de 4 puntos) **es el mismo para todos los grupos**, lo cual significa que independientemente del profesor de prácticas, cada estudiante es responsable de comprender todo lo explicado en estos **guiones de prácticas (que son comunes a todos los grupos y son materia de examen)** y desarrollar las habilidades que se ejercitan en los mismos (que también son materia de examen).

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Para el caso concreto de la Actividad 4.2 de esta práctica, si hay interés por parte de los estudiantes, si la temporización lo permite, y si los profesores perciben que hay sincronización entre los distintos grupos de los distintos profesores, podría establecerse una fase adicional de desactivación global (entre todos los grupos) de las bombas más difíciles de cada grupo, entendiendo por difícil que haya sido poco escogida o, que aunque haya sido escogida, no haya sido desactivada.

6 Referencias

[1] Manuales de Intel sobre IA-32 e Intel64, en concreto el volumen 2: "Instruction Set Reference"
https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf

[2] WikiBooks, X86_Disassembly http://en.wikibooks.org/wiki/Special:Search/X86_Disassembly/
Analysis Tools http://en.wikibooks.org/wiki/X86_Disassembly/Analysis_Tools
Disassemblers and Decompilers http://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompilers

[3] GNU binutils, artículo Wikipedia: http://en.wikipedia.org/wiki/GNU_Binutils
Sitio web http://www.gnu.org/software/binutils/
Manuales (incluye gas, ld, nm…) http://sourceware.org/binutils/docs/
Documentación objdump https://sourceware.org/binutils/docs/binutils/objdump.html

[4] GNU Debugger, GDB, Wikipedia http://en.wikipedia.org/wiki/Gdb
Sitio web http://www.gnu.org/s/gdb/
Manuales http://sourceware.org/gdb/current/onlinedocs/gdb/
Chuletario http://www.csd.uoc.gr/~hy255/refcards/gdb-refcard.pdf
Resumen comandos http://web.cecs.pdx.edu/~jrb/cs201/lectures/handouts/gdbcomm.txt

[5] GDB –TUI Tutorial http://beej.us/guide/bggdb/

[6] Data Display Debugger, DDD, Wiki http://en.wikipedia.org/wiki/Data_Display_Debugger
Sitio web http://www.gnu.org/s/ddd/
Manuales (incluye tutorial) http://www.gnu.org/s/ddd/manual/html_mono/ddd.html
http://www.gnu.org/s/ddd/manual/html_mono/ddd.html#Sample%20Session

[7] Paquete ddd para Ubuntu 18.04 https://packages.ubuntu.com/bionic/ddd
Buscador de paquetes http://packages.ubuntu.com/

[8] Depuración de binarios *stripped* https://reverseengineering.stackexchange.com/questions/1935/

[9] Depuración de binarios *PIE* https://reverseengineering.stackexchange.com/questions/8724/

[10] GDB, scripts de inicialización https://sourceware.org/gdb/onlinedocs/gdb/objfile_002dgdbdotext-file.html

[11] Wikipedia, Editor hex http://en.wikipedia.org/wiki/Hex_editor
Comparación de editores hex http://en.wikipedia.org/wiki/Comparison_of_hex_editors

[12] Google, "Ubuntu hex editors" http://www.google.com/search?q=ubuntu+hex+editors
Ubuntu 18.04, paquete ghex https://packages.ubuntu.com/bionic/ghex
Ubuntu 18.04, paquete hexedit https://packages.ubuntu.com/bionic/hexedit
Ubuntu 18.04, ncurses-hexedit https://packages.ubuntu.com/bionic/ncurses-hexedit
Blog UnixLab, 5 editores http://unixlab.blogspot.com/2009/08/five-gui-hex-editors-for-ubuntu.html
wxHexEditor (Linux/Win) http://sourceforge.net/projects/wxhexeditor/

[13] Apuntes y presentaciones de clase
Libro CS:APP: Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 3rd Ed., Pearson, 2016. http://csapp.cs.cmu.edu/

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND

Apéndice 1. Ejemplo de explicación mediante *script gdb*

A continuación se muestra un ejemplo del tipo de *script* de explicación deseado, en donde se comenta profusamente por qué se tomó cada decisión, de forma que el lector tenga una idea clara de cómo hizo el autor para ir añadiendo una a una las instrucciones *gdb* utilizadas, cuyo resultado final ha sido no sólo imprimir por pantalla las claves originales, sino además cambiar al menos un carácter de la contraseña y un dígito del pin.

Naturalmente, se valora la brevedad del script, ya que se supone que los compañeros deberían ser capaces de desarrollarlo por sí mismos en un plazo de tiempo de unos 15-60min. También se valora la claridad de las explicaciones. Tener en cuenta que el árbitro definitivo de si la explicación es válida o no es el estudiante que reclama, después de haberle dedicado más de media hora a la bomba, y por tanto con menos de media hora para resolver otra bomba que se le asigne, así que conviene que la explicación sea indiscutiblemente sencilla de entender.

```
# Practica 4, Actividad 4.1: fichero explain.gdb, explicacion de la bomba

# CONTRASEÑA: abracadabra
#       PIN: 7777

# MODIFICADA: hola,adios.
#       PIN: 123

# Describe el proceso logico seguido
# primero: para descubrir las claves, y
# despues: para cambiarlas

# Pensado para ejecutar mediante "source explain.gdb"
# o desde linea de comandos con gdb -q -x explain.gdb
# Renombrar temporalmente el fichero "bomba-gdb.gdb"
# para que no se cargue automat. al hacer "file bomba"

# funciona sobre la bomba original, para recompilarla
# usar la orden gcc en la primera linea de bomba.c
# gcc -Og bomba.c -o bomba -no-pie -fno-guess-branch-probability

############################################################

### cargar el programa
    file bomba
### util para la sesion interactiva, no para source/gdb -q -x
#   layout asm
#   layout regs
### arrancar programa, notar automatizacion para teclear hola y 123
    br main
    run < <(echo -e hola\\n123\\n)
### hicimos ni hasta call boom, antes pide contraseña y tecleamos hola
### si entramos en boom explota y hay que empezar de nuevo
### la decision se toma antes, justo antes de call boom
### hay un je que se salta la bomba, y el test anterior
### activaria ZF si el retorno de strncmp produjera 0,
### es decir, si 0==strncmp(rdi,rsi,edx)
#   0x4007b6 <main+91>  lea   0x30(%rsp),%rdi
#   0x4007bb <main+96>  mov   $0xd,%edx
#   0x4007c0 <main+101> lea   0x2008a1(%rip),%rsi # 0x601068 <password>
#   0x4007c7 <main+108> callq 0x4005d0 <strncmp@plt>
#   0x4007cc <main+113> test  %eax,%eax
#   0x4007ce <main+115> je    0x4007d5 <main+122>
#   0x4007d0 <main+117> callq 0x400727 <boom>
#   0x4007d5 <main+122> lea   ...
### avancemos hasta strncmp para consultar los valores
    br *main+108
    cont
### escribir "hola" cuando pida contraseña, resuelto ya en run
### ahora mismo estamos viendo de donde sale la contraseña
#   0x4007c0 <main+101> lea 0x2008a1(%rip),%rsi # 0x601068 <password>
### imprimir la contraseña y recordar que esta en 0x601068 longitud 13B
#   p(char*    )$rsi
#   p(char*    )0x601068
    p(char[0xd])password
### dejar que strncmp salga mal y corregir eax=0 para evitar boom()
    ni
```


Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/

BY-NC-ND

```
    set $eax=0
    ni
    ni
### siguiente bomba es por tiempo
#   0x4007df <main+132> callq  0x4005f0 <gettimeofday@plt>
#   0x4007e4 <main+137> mov    0x20(%rsp),%rax
#   0x4007e9 <main+142> sub    0x10(%rsp),%rax
#   0x4007ee <main+147> cmp    $0x5,%rax
#   0x4007f2 <main+151> jle    0x4007f9 <main+158>
#   0x4007f4 <main+153> callq  0x400727 <boom>
#   0x4007f9 <main+158> lea    ...
### avanzar hasta el cmp
    br *main+147
    cont
### falsear tiempo=0 por si acaso se ha tardado en teclear
    set $eax=0
    ni
    ni
### siguiente bomba compara resultado scanf con variable de memoria
#   0x400841 <main+230> mov   0x200819(%rip),%eax # 0x601060 <passcode>
#   0x400847 <main+236> cmp   %eax,0xc(%rsp)
#   0x40084b <main+240> je    0x400852 <main+247>
#   0x40084d <main+242> callq 0x400727 <boom>
#   0x400852 <main+247> lea ...
### avanzar hasta cmp para consultar valores
    br *main+236
    cont
### escribir "123" cuando pida pin, resuelto ya en run
### imprimir el pin y recordar que esta en 0x601060 tipo int
#   p*(int*)0x601060
    p (int )passcode
### corregir sobre la marcha EAX para que cmp salga bien
    set $eax=123
    ni
    ni
### siguiente bomba es por tiempo
#   0x40085c <main+257> callq  0x4005f0 <gettimeofday@plt>
#   0x400861 <main+262> mov    0x10(%rsp),%rax
#   0x400866 <main+267> sub    0x20(%rsp),%rax
#   0x40086b <main+272> cmp    $0x5,%rax
#   0x40086f <main+276> jle    0x400876 <main+283>
#   0x400871 <main+278> callq  0x400727 <boom>
#   0x400876 <main+283> callq  0x400741 <defused>
#   0x40087b            nopl
### avanzar hasta el cmp
    br *main+272
    cont
### falsear tiempo=0 por si acaso se ha tardado en teclear
    set $eax=0
    ni
    ni
### hemos llegado a defused, fin del programa
    ni

############################################################

### permitir escribir en el ejecutable
    set write on
### reabrir ejecutable con permisos r/w
    file bomba
### realizar los cambios
    set {char[13]}0x601068="hola,adios.\n"
    set {int     }0x601060=123
### comprobar las instrucciones cambiadas
    p  (char[0xd])password
    p  (int      )passcode
### salir para desbloquear el ejecutable
    quit

############################################################
```

Figura 6: ejemplo de script de explicación, para ejecutar con gdb –q –x explain.gdb

Este trabajo está licenciado bajo Creative Commons Attribution-NonCommercial-NoDerivatives 4.0.
Para ver una copia de esta licencia, visitar http://creativecommons.org/licenses/by-nc-nd/4.0/
BY-NC-ND