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 consultando y ejecutando su código binario sin información de depuración.
- 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/g++`. Para desensamblar se pueden usar desensambladores (por ejemplo, `objdump` [2] [3]) o depuradores (por ejemplo, ddd [4] [5], recomendando ddd-3.3.12-2.1 o superior para gcc-4.4.3). Los depuradores generalmente incorporan un desensamblador.

Con ddd se puede ejecutar paso a paso un programa aunque no se haya compilado con información de depuración (es decir, no se usó la opción `-g` de `gcc/g++`) usando la ventana de código máquina (*View→Machine Code Window*) de ddd. Si no aparece el código en la ventana de código máquina, probar instalando alguna versión más reciente de ddd [5], y/o ejecutando en la línea de comandos gdb:

```
info line main      ó      info line nombre_función
disas     main      ó      disas     nombre_función
```

De esta forma aparecerá en la ventana de código máquina o en la de comandos el código de la función que se indique. Para visualizar el código en un rango de direcciones se puede escribir en la ventana de comandos (consola) gdb:

```
disas [ primera_dirección [[,] última_dirección] ]
```

Según la versión de gcc, puede ser necesario separar la primera y última dirección con ",", para no obtener el error "*A syntax error in expression, near <addr>*". Las direcciones se pueden obtener, por ejemplo, con `objdump`. Si sale ese error siempre y se necesita usar `disas`, seguramente interesará tener visible (en un terminal o editor al lado de la ventada del depurador) el desensamblado del fichero ejecutable. De todas formas, se recomienda usar una versión de ddd adecuada a la de gcc, para que la ventana de código máquina funcione con normalidad. También se recomienda usar `help` en la consola gdb y/o consultar la documentación del depurador [6]. Recordar que el modo gráfico ddd es fácil, bonito e intuitivo, pero no tan potente como el modo texto de gdb.

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 ddd la opción Data→Memory permite visualizar en la ventana de datos el contenido de memoria a partir de una dirección dada; se pueden especificar distintos formatos (*string*, *char*, *integer*, *instruction*, etc.)

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 [7]. 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 `ghex2`). En el portátil con Ubuntu, se puede uno instalar *ghex*, o *hexedit*, o *ncurses-hexedit* (y ejecutar `ghex2`, `hexedit`, o `hexeditor`). Otra opción es descargar e instalar *wxHexEditor* en Ubuntu o Windows [8].

3 Ejemplo tutorial

Para motivar el uso del desensamblador y del editor hexadecimal, proponemos el ejercicio de la *bomba digital*: un programa que solicita una contraseña, 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 comprueba la contraseña, 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 código, y el máximo tiempo que se puede tardar en cada entrada de datos.

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

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

void boom(){
       printf("***************\n");
       printf("*** BOOM!!! ***\n");
       printf("***************\n");
       exit(-1);
}
void defused(){
       printf("·························\n");
       printf("··· bomba desactivada ···\n");
       printf("·························\n");
       exit(0);
}

int main(){
#define SIZE 100
       char pass[SIZE];
       int  pasv;

#define TLIM 5
       struct timeval tv1,tv2;       // gettimeofday() secs-usecs
       gettimeofday(&tv1,NULL);

       printf("Introduce la contraseña: ");
       fgets(pass,SIZE,stdin);
       if (strncmp(pass,password,strlen(password)))
           boom();

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

       printf("Introduce el código: ");
       scanf("%i",&pasv);
       if (pasv!=passcode) boom();

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

       defused();
}
```

Figura 1: bomba.c: bomba digital con 2 fases: *string* y enteros

Comprobar el correcto funcionamiento de todas las ramas de control (sentencias *if*), probando a:

- Introducir una contraseña incorrecta (1er *if*)
- tardar mucho en ese primer paso (siendo la contraseña *correcta*, 2º *if*)
- introducir un código incorrecto
- tardar mucho en introducir el código *correcto*

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 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 (`gcc -m32 bomba.c -o bomba`), y ejecutarlo paso a paso con ddd para descubrir las claves. Probablemente sigamos un proceso deductivo similar a éste:

- Poner un *breakpoint* en la primera instrucción y empezar la ejecución. Si no se puede visualizar la *Machine Code Window* (ni siquiera con `info line main`) probablemente deseemos indicar `break main` ó `break *<addr>` en la consola gdb, y tener un desensamblado visible en una ventana al lado del depurador, para consultar cómodamente las direcciones del ejecutable. El resto de esta descripción supone que se puede usar la *Machine Code Window*.
- Ir avanzando con *Nexti* (para no meternos en subrutinas). Notar que aunque compilamos sin información de depuración, podemos ver los nombres de las funciones invocadas por la bomba, lo cual nos permite ir haciéndonos una idea de "por dónde van los tiros".
- En algún momento se nos pedirá la contraseña. Introducir algún valor que podamos reconocer después, como por ejemplo "`kip`".
- Seguir avanzando con *Nexti*. La bomba explota, en este caso dentro de una función llamada `boom`. Poner un *breakpoint* en dicha llamada para futuras ejecuciones.
- Volver a ejecutar, aprovechando la parada en el primer *breakpoint* para eliminarlo (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 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 el siguiente ejercicio 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` y modificando entonces el valor de EAX en línea de comandos gdb (`set $eax=0`). Podemos dejar el *breakpoint* de `test` y eliminar el de `call boom` 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` en línea de comandos gdb (o con el botón *Set*) antes de ejecutar `cmp`.
- Pasado este segundo problema, en algún momento se nos pedirá el código. Introducir algún valor que podamos reconocer después, como por ejemplo "`1945`".
- Con la experiencia adquirida, poco después comprobamos que hubiéramos debido introducir `7777`. Añadimos un *breakpoint* en la instrucción `cmp %eax,%edx`, para poder modificar también en ejecuciones futuras `set $edx=7777` 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.
- Por último se invoca `defused` y comprobamos que 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 algún error 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 salvado de sesiones ddd o ficheros de inicio 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 con ddd.

```
...
0x080486e8 <main+120>:       call   0x8048524 <strncmp@plt>
0x080486ed <main+125>:       test   %eax,%eax
0x080486ef <main+127>:       je     0x80486f6 <main+134>
0x080486f1 <main+129>:       call   0x8048604 <boom>
0x080486f6 <main+134>:       ...
...
0x08048705 <main+149>:       call   0x80484c4 <gettimeofday@plt>
...
0x08048718 <main+168>:       cmp    $0x5,%eax
0x0804871b <main+171>:       jle    0x8048722 <main+178>
0x0804871d <main+173>:       call   0x8048604 <boom>
0x08048722 <main+178>:       ...
...
0x0804873f <main+207>:       call   0x8048504 <__isoc99_scanf@plt>
0x08048744 <main+212>:       mov    0x24(%esp),%edx
0x08048748 <main+216>:       mov    0x804a044,%eax
0x0804874d <main+221>:       cmp    %eax,%edx
0x0804874f <main+223>:       je     0x8048756 <main+230>
0x08048751 <main+225>:       call   0x8048604 <boom>
0x08048756 <main+230>:       ...
...
0x08048765 <main+245>:       call   0x80484c4 <gettimeofday@plt>
...
0x08048778 <main+264>:       cmp    $0x5,%eax
0x0804877b <main+267>:       jle    0x8048782 <main+274>
0x0804877d <main+269>:       call   0x8048604 <boom>
0x08048782 <main+274>:       call   0x804863a <defused>
...
```

Figura 2: ventana de código máquina 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 un salto hacia adelante de 5 posiciones, las que ocupa la llamada a *boom*.

```
...
80486ef:    74 05                jle    8048722 <main+0xb2>
80486f1:    e8 0e ff ff ff       call   8048604 <boom>
...
804871b:    7e 05                jle    8048722 <main+0xb2>
804871d:    e8 e2 fe ff ff       call   8048604 <boom>
...
804874f:    74 05                je     8048756 <main+0xe6>
8048751:    e8 ae fe ff ff       call   8048604 <boom>
...
804877b:    7e 05                jle    8048782 <main+0x112>
804877d:    e8 82 fe ff ff       call   8048604 <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. Este es otro ejemplo de manipulación que se puede conseguir con el modo texto de gdb, y no con el modo gráfico de ddd. Los comandos necesarios son:

- `set write on` `# para permitir escribir en el ejecutable`
- `file bomba` `# reabrir el ejecutable con nuevos permisos r/w`
- `info line main` `# visualizar Machine Code Window, localizar boom`
- `set *(char*)0x080486ef=0xeb` `# cambiar el primer je por jmp`
- `set *(char*)0x0804874f=0xeb` `# cambiar el segundo je por jmp`
- `set write off` `# dejar la opción de escritura desactivada`
- `file <otro prog>` `# para evitar "text file busy"`
- `quit` `# salir garantiza que no bloqueamos el ejecutable`

Ejercicio 3: editor hexadecimal

Se puede ejecutar en un terminal la bomba modificada en el ejercicio anterior para comprobar que efectivamente no explota. De todas formas, es mucho más cómodo utilizar un editor hex para modificar (incluso localizar) los códigos de operación. Recompilar la bomba (`gcc –m32 bomba.c -o bomba`), y ejecutar `ghex2 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 0e ff ff ff`. 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 ae fe ff ff`.

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 punto 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 `0x1c(%esp)`.
- El argumento de `printf()` es `0x080488d4`, *string* pidiendo la contraseña (comprobarlo volcándolo con Data→Memory).
- `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 `0x28(%esp)`.
- `strlen()` calcula la longitud del argumento *string*, y al volcar la dirección `0x0804a034` acabamos de encontrar la contraseña, que para mayor claridad aparece etiquetada como *password*. Continuando el estudio, vemos que este cálculo de la longitud se usa como tercer argumento de `strncmp()`. Si esa cadena y la que se leyó de teclado son iguales se salta la llamada a la bomba (`test/je/call`).
- Un estudio similar del código que sigue y la página de manual de `gettimeofday()` nos desvela que haber tardado más de 5 segundos desde el inicio del programa causa llamada a la bomba.
- Más adelante se comparan EAX, traido de `0x0804a044` (que para más claridad aparece etiquetado como *passcode* al volcarlo con Data→Memory) con EDX, traido de `0x24(%esp)`, 2º argumento de `scanf()`. Hemos encontrado la segunda clave.
- El estudio del final del programa nos revela el mismo límite de 5 segundos para esta segunda parte.

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, es inmediato modificarlas, con el propio depurador o con el editor hexadecimal. 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/C++ que simule la explosión de una bomba si el usuario no introduce correctamente dos claves o si las introduce en un tiempo superior a 1 minuto. No es necesario que la bomba explote mientras el usuario está tecleando las claves; basta con que lo haga después de introducir la 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 código (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 programa se compilará sin información de depuración y sin librerías (salvo libC, incluida implícitamente), sin aplicarle opciones o utilidades que eliminen más símbolos. Se permitirá utilizar optimización de nivel 2 como máximo, de manera que la orden de compilación más compleja será `gcc -m32 –O2 bomba.c -o bomba`.

No se aplicarán algoritmos con cálculos irreversibles de tipo criptográfico. Las claves deben ser valores fijos que no dependan de la fecha, la hora del día, el tiempo que se tarde en teclearlas, etc. Por supuesto que no deben ser valores generados aleatoriamente. Los profesores serán los árbitros definitivos sobre qué trucos valen y cuáles no.

Sería válido, por ejemplo, usar la cifra del César para codificar la contraseña alfanumérica, o usar indexación en un array para transformar el código numérico. Quedaría a juicio de los profesores decidir por ejemplo si es válido usar la sucesión de Fibonacci para codificar el código numérico, 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 poder participar en la siguiente fase (desactivación), se debe **entregar** al profesor, en SWAD→ Evaluacion→ Mis Trabajos, en la fecha que se indique:

1. El **ejecutable** generado (sin información de depuración, sin eliminar símbolos con utilidades)
2. El **fuente** C/C++, indicando en un comentario en la cabecera la **orden de compilación usada**
3. **Explicación** del método preferido para desactivar la bomba. La explicación debería ser comprensible para cualquier compañero de clase que domine la materia explicada en la asignatura. Las explicaciones las verán 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 las claves** en el tiempo estipulado (entre 15 minutos y una hora). Eventualmente, si algún compañero reclama en la siguiente fase (desactivación) que una bomba no se puede desactivar, se consultará 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.

El **formato** recomendado para las explicaciones consiste en indicar, para cada decisión tomada durante la sesión de depuración, la siguiente información:

a. Volcado **ensamblador** de dónde estamos parados (*copy-paste* de *Machine Code Window*)…
   i. mostrando dirección del *breakpoint*, o…
   ii. explicando cómo y por qué se llegó a la instrucción actual
b. Volcado *Data→Memory* o *Status→Registers* de la **información** relevante
c. **Razonamiento** de qué se hace a continuación (por qué se toma qué decisión)

El **ejecutable** (sin información de depuración y en un .zip) también se debe depositar en la **Zona Común** del grupo de prácticas (Asignatura→ Zona Común→ seleccionar grupo), para que lo vean los compañeros. Notar que cuando se selecciona la Zona Común del grupo, deja de verse la Zona Común de la asignatura, y viceversa.

4.2 Desactivar bombas digitales

Se trata de desactivar un par de bombas digitales realizadas por otros compañeros (que también la hayan entregado en la fecha establecida). Para la desactivación sólo se dispondrá del **ejecutable** del programa, que estará depositado en la **Zona Común** 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. Cada estudiante desactivará dos bombas digitales (distintas de la suya). Recordar que hay caracteres codificados con más de un byte, consultar la tabla de códigos UTF-8 en la Wikipedia (o buscarla con Google).

Para poder participar en la última fase, se debe **entregar** al profesor, en SWAD→ Evaluacion→ Mis Trabajos, en la fecha que se indique:

1. Identificación de las bombas escogidas (de qué compañeros son)
2. El nombre de las herramientas utilizadas (depuradores, desensambladores…)
3. **Explicación** del método seguido para localizar o descubrir las claves. Esta explicación debe permitir al profesor deducir que las claves las ha encontrado uno mismo

Se recomienda usar para estas explicaciones el mismo formato usado para el apartado anterior en la explicación de la propia bomba. Naturalmente, no es válido indicar solamente la solución (claves) sin aclarar cómo se ha obtenido.

4.3 Modificar bombas digitales

Se trata de modificar las bombas desactivadas anteriormente para que les exploten a sus autores, usando para ello un depurador o editor hexadecimal, probablemente. Las bombas modificadas se pueden mostrar a los compañeros que las han montado para que comprueben que les explotan al ejecutarlas con las claves antiguas.

Al profesor se le debe **entregar** en SWAD→ Evaluacion→ Mis Trabajos la siguiente información:

1. Bombas **modificadas**, y su identificación (de qué compañeros eran las originales)
2. El nombre de las herramientas utilizadas (depuradores, editores hexadecimal…)
3. **Explicación** de lo que se ha hecho para modificar el código. Esta explicación debe permitir al profesor deducir que la modificación en el código ejecutable la ha realizado uno mismo.
4. Comentarios que se le ocurran a uno sobre qué podría hacer para aumentar la protección de su código.

El formato de la explicación es más libre en este caso. No se espera una modificación masiva del código máquina, convirtiendo la bomba en otra totalmente distinta, sino modificaciones puntuales, posiblemente afectando sólo a los datos del programa, para conseguir cambiar las claves. De nuevo, en caso de duda los profesores arbitrarán si una modificación concreta es válida o no.

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.

Por ejemplo, puede que en un grupo sólo se deban entregar los fuentes de las bombas y las explicaciones de las distintas fases, encargándose el profesor de compilar las bombas y asignarlas para la segunda fase alrededor del puente de Diciembre.

Puede que en otro grupo se pueda trabajar y entregar por parejas, pero que el profesor de prácticas anote las asistencias al laboratorio, permitiendo que sólo los que asisten a todas las sesiones participen en las distintas fases. La elección de bombas a desactivar podría ser voluntaria por parte de la pareja.

Para el caso concreto 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 distintos grupos, podría establecerse una fase adicional de desactivación 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"
http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-2a-2b-instruction-set-a-z-manual.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 http://sourceware.org/binutils/docs-2.22/binutils/objdump.html#objdump

[4] 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

[5] Ubuntu, paquete ddd actualizado http://launchpadlibrarian.net/76089830/ddd_3.3.12-2.1_amd64.deb
Buscador de paquetes http://packages.ubuntu.com/

[6] 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

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

[8] Google, "Ubuntu hex editors" http://www.google.com/search?q=ubuntu+hex+editors
Ubuntu, paquete ghex http://manpages.ubuntu.com/manpages/lucid/en/man1/ghex2.1.html
Ubuntu, paquete hexedit http://manpages.ubuntu.com/manpages/lucid/en/man1/hexedit.1.html
Ubuntu, paquete ncurses-hexedit http://manpages.ubuntu.com/manpages/lucid/en/man1/hexeditor.1.html
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/

[9] Apuntes y presentaciones de clase
Libro CS:APP: Randal E. Bryant, David R. O'Hallaron: "Computer Systems: A Programmer's Perspective", 2nd Ed., Pearson, 2011. http://csapp.cs.cmu.edu/