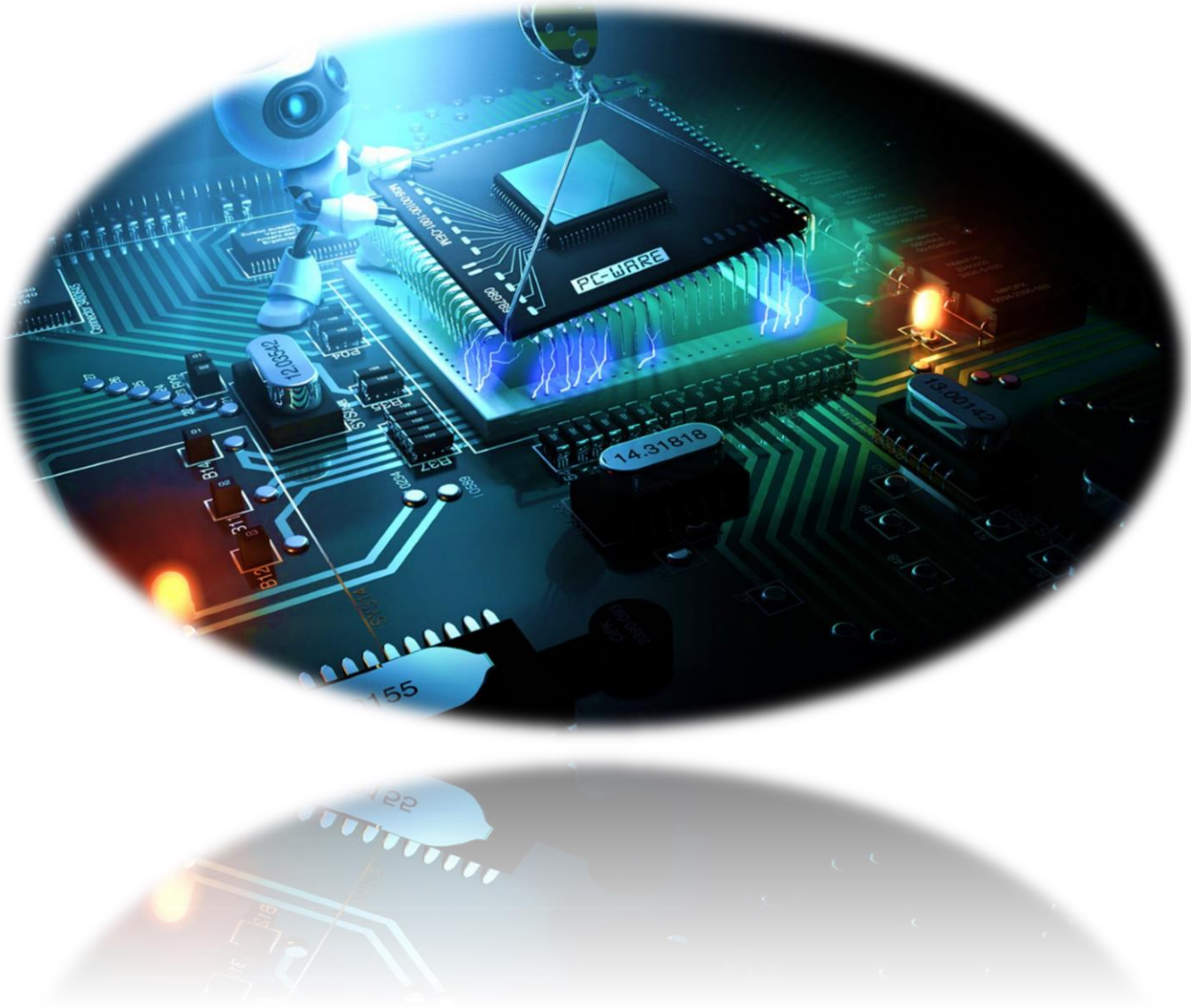
# PRÁCTICA 4

ESTRUCTURA DE COMPUTADORES




BRYAN MORENO PICAMÁN

# Contenido

# Diario de trabajo

A continuación se detalla un diario de trabajo con los días dedicados a la práctica y las partes que se han desarrollado:

- 29 noviembre.- Primera lectura del tutorial de prácticas, y realización de pruebas que se indican.
- 4 diciembre.- Comienzo de los códigos de la práctica.
- 5 diciembre.- Auto resolución de la bomba, pruebas con DDD y documentación.

Nota: Ordenador usado es:

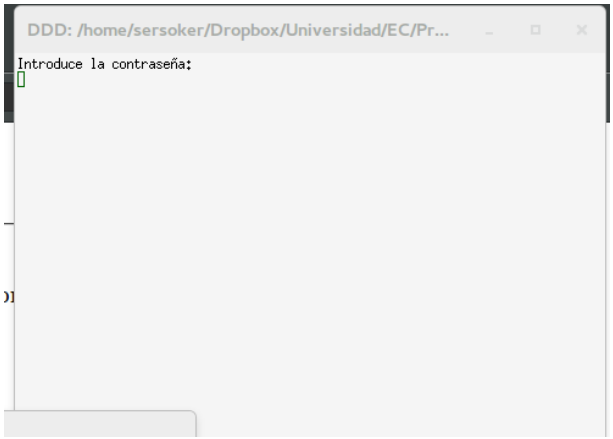
Intel(R) Core(TM) i5-4210H CPU @ 2.90GHz, 8Gb RAM, 1TB HDD.

Las órdenes de compilación usadas han sido:

g++ -m32 archivo.c –o archivo

# Bomba Bryan Moreno Picamán

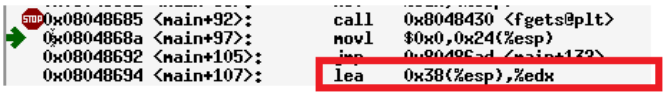
Inicialmente cuando se lanza el programa obtenemos una ventana que pide una

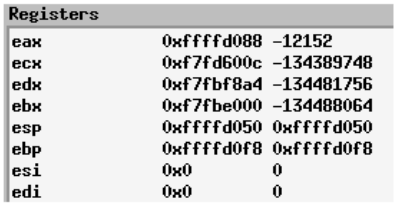


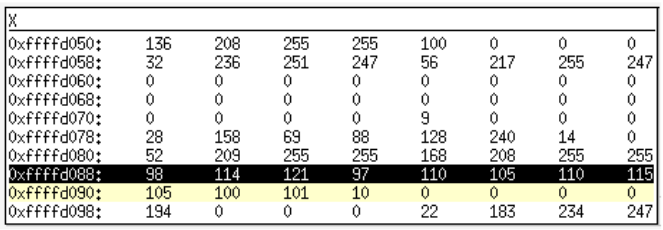

Así que empezaremos a averiguar esta contraseña y los pasos seguidos.

## Paso 1.

Lanzar el programa y ver donde se almacenan los datos, este paso es importante porque así vemos donde se almacena la contraseña que metemos y así vemos cuando se usan estos datos y para qué.

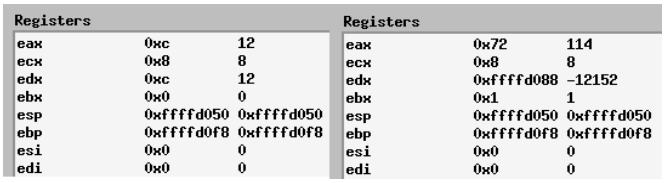



Después de la llamada a fgets, que es la que recoge los datos desde teclado, vemos que las siguientes operaciones se realizan usando como puntero el registro esp (se puede ver que hay un bucle de x iteraciones, siendo x el número de letras insertadas por teclado).

Aquí podemos observar que ebx aumenta de 1 en 1 entre iteraciones del bucle (capruta de 1º y 2º iteración) por lo que después de observar el código se ve claramente que se usa como índice del bucle.



## Paso 2.

Ya tenemos los datos y el índice del bucle localizado, el siguiente paso es averiguar que hace el bucle exactamente.

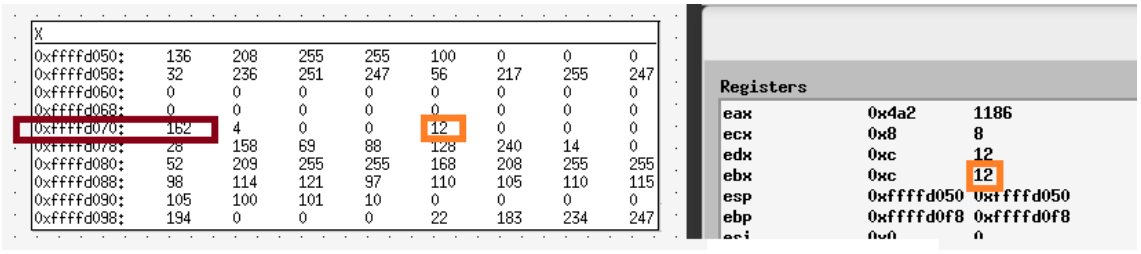
```
0x08048692 <main+105>:    jmp    0x80486ad <main+132>
0x08048694 <main+107>:    lea    0x38(%esp),%edx
0x08048698 <main+111>:    mov    0x24(%esp),%eax
0x0804869c <main+115>:    add    %edx,%eax
0x0804869e <main+117>:    movzbl (%eax),%eax
0x080486a1 <main+120>:    movsbl %al,%eax
0x080486a4 <main+123>:    add    %eax,0x20(%esp)
0x080486a8 <main+127>:    addl   $0x1,0x24(%esp)
0x080486ad <main+132>:    mov    0x24(%esp),%ebx
0x080486b1 <main+136>:    lea    0x38(%esp),%eax
0x080486b5 <main+140>:    mov    %eax,(%esp)
0x080486b8 <main+143>:    call   0x80484a0 <strlen@plt>
0x080486bd <main+148>:    cmp    %eax,%ebx
0x080486bf <main+150>:    jb     0x8048694 <main+107>
0x080486c1 <main+152>:    mov    0x804a03c,%eax
0x080486c6 <main+157>:    cmp    %eax,0x20(%esp)
```

Analizando el bucle, tenemos que leemos un valor con lea en edx, se pone eax al valor que contiene 0x24(%esp), que observando el bucle vemos que coincide con el valor del incide del bucle, es más, observando más de cerca primero se le da valor a 0x24(%esp) y este es copiado posteriormente a ebx, que se usa luego para la condición del bucle.

El paso anterior es muy importante, ya que el primer mov, hace uso de este valor para llevar a %eax el valor que tiene que usar en esa iteración (calculando el desplazamiento con movzbl.

Después de esto, ya tenemos en %eax el nuevo valor a usar, y en 0x38(%esp) el valor acumulado de las operaciones anteriores (inicialmente 0), este bucle continua haciéndose hasta que se “agotan” todos los valores.

En esta captura podemos observar que tenemos por un lado el resultado acumulado (rojo) y por el otro los índices, tanto en memoria como en registro (naranja).

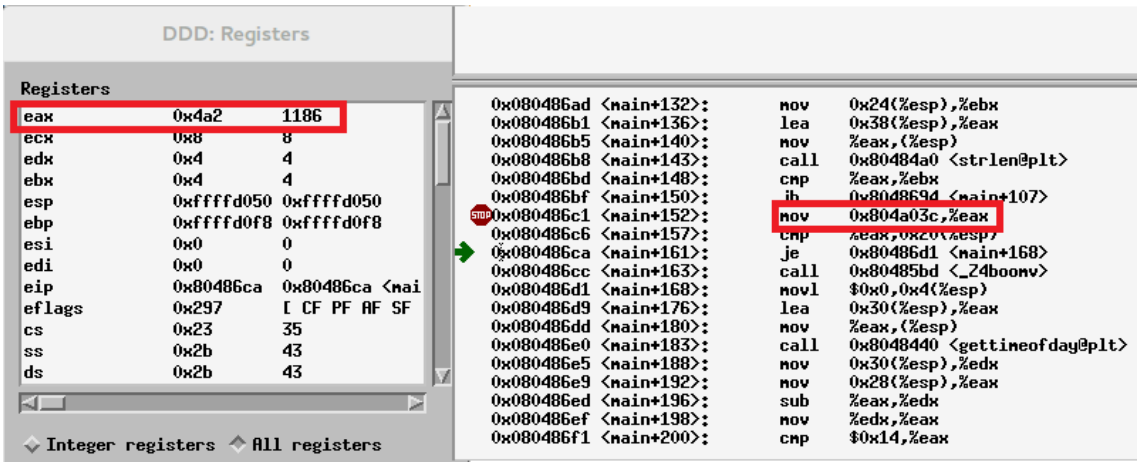


```
(gdb) x/40b 0xffffd050
0xffffd050:  136  208  255  255  100  0    0    0
0xffffd058:  32   236  251  247  56   217  255  247
0xffffd060:  0    0    0    0    0    0    0    0
0xffffd068:  0    0    0    0    0    0    0    0
0xffffd070:  152  4    0    0    12   0    0    0
0xffffd078:  28   158  69   88   128  240  14   0
0xffffd080:  52   209  255  255  168  208  255  255
0xffffd088:  98   114  121  97   110  105  110  115
0xffffd090:  105  100  101  10   0    0    0    0
0xffffd098:  194  0    0    0    22   183  234  247
```

```
Registers
eax            0x4a2      1186
ecx            0x8        8
edx            0xc        12
ebx            0xc        12
esp            0xffffd050 0xffffd050
ebp            0xffffd0f8 0xffffd0f8
esi            0x0        0
```

## Paso 3.

Una vez se sabe que se hace con los valores (en este caso una suma a una posición de memoria), tenemos que ver que se hace con estos datos, si continuamos observando el código (después del bucle anterior), llegamos a una parte del código que saca algo de memoria a registro y lo compara directamente con el valor acumulado de la operación anterior, este es el momento “clave”.




Aquí ya vemos que el valor “cifrado” es 1186, es decir la clave que se inserte debe darnos el mismo resultado que este. Hay que tener en cuenta que en memoria el valor no se almacena de forma normal, sino que está guardado en complemento a 2, por lo que si examinamos el valor de memoria de 0x20(%esp) no coincidirá con el que realmente tenemos que meter.

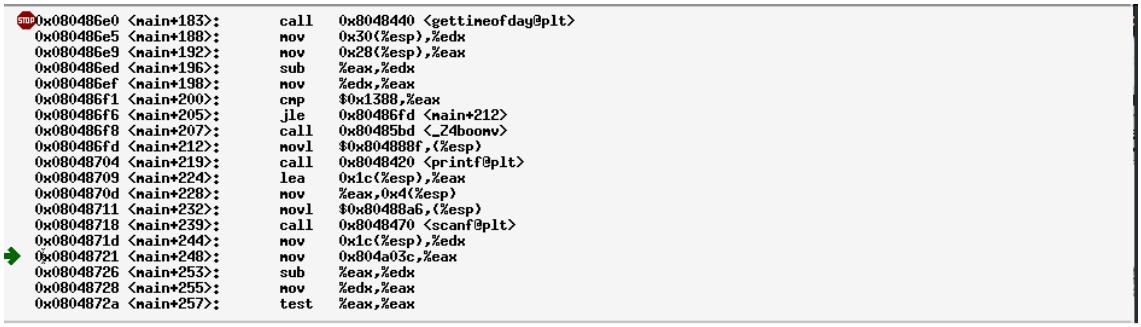
Para probar esto se necesita meter un valor de “letras” cuya suma de caracteres en valor ASCII decimal sume exactamente 1186, de la misma forma siendo esto un “cifrado” simple, no solo admite 1 contraseña, si no que pueden darse varias que permitan pasar al siguiente paso, esto se ha hecho para intentar confundir un poco al usuario que intente “romper” la bomba y tenga que entender exactamente que se almacena y por qué tanto en memoria como en los registros.

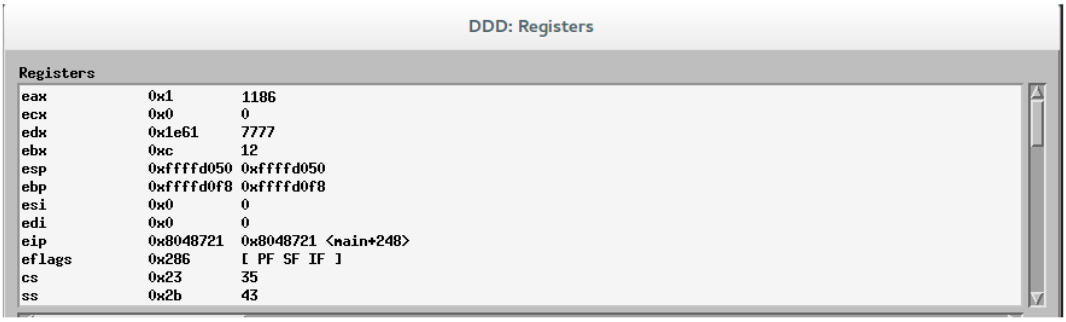
Después de este paso pasamos a la clave numérica.

## Paso 4.

En este paso vamos a localizar donde se ve esta clave y que espera que se introduzca, al igual que en los pasos anteriores con capturas de memoria y registros para que sea más sencillo.

Inicialmente con la contraseña anterior averiguada, introducimos una clave errónea o al azar para ver que realiza el programa con ella, vemos claramente que después de la llamada a scanf, la clave introducida (7777) se guarda en edx, y al igual que en el caso anterior se pasa un dato de memoria y se compara con el introducido, en la captura podemos observar como no coindicen por lo que no funcionaria.




Vemos que en este caso el valor que se debe introducir coincide en dirección y valor con el del caso anterior, es decir solo utilizamos una variable para almacenar ambos valores, esto se ha hecho con vistas a que la gente no aprenda de forma mecánica que registros se usan para comparar y tenga que ver el código y comprenderlo.

## Resumen.

Así pues como resumen de modificaciones nos e ha querido realizar un programa complejo, pero si modificarlo lo suficiente para que se tenga que entender que hace y porque, las partes modificadas han sido:

- Paso a valor numérico de cada uno de los caracteres introducidos por pantalla y sumados en una sola variable, añadiendo para esta función un bucle.
- Añadida complicación “extra” no existe una única clave valida, por lo que si el usuario no sabe qué hace exactamente puede llegar a confundirse si con varias contraseñas obtiene una desactivación de la bomba.
- Paso de 2 variables de clave a 1 sola, en una usa su valor cifrado (suma de valores numéricos) y en la otra su valor numérico como tal, usado para confundir al usuario y evitar posibles resoluciones “mecánicas” de la bomba.

Datos:

Contraseña: bryaninside

*Nota: Hay varias validas, pero esta es la que he usado yo personalmente

Clave: 1186

*Nota: Suma de los valores ascii de cada una de las letras.

Estimación de tiempo de resolución:

- Si se comprenden bien bucles y no se intenta equipararla a la bomba de ejemplo: 15/20min máximo.
- Otros casos: 30/50min máximo.