

Buscar y modificar claves de la bomba usando DDD(gdb) y GHex

Por: Antonio Jesús Heredia Castillo

1 Clave Alfanumérica

1.1 Búsqueda de clave alfanumérica.

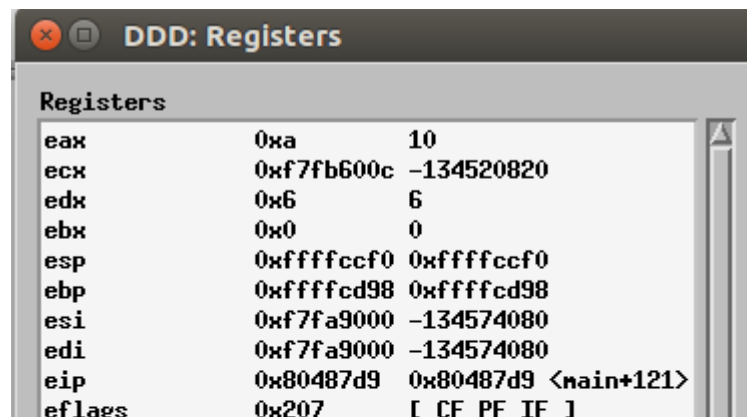
Lo primero que realizamos es buscar donde se encuentra la clave alfanumérica encriptada guardada.

Para ello nos ayudaremos primero de **DDD**. Cargamos nuestro ejecutable y vemos el código ensamblador

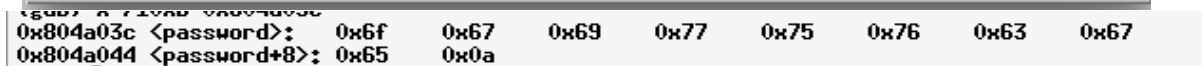
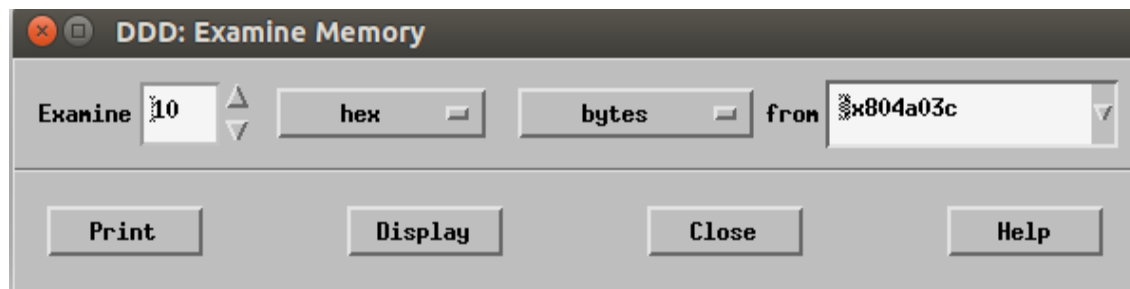
```
0x08048760 <nain+0>: lea    0x4(%esp),%ecx
0x08048764 <nain+4>: and    $0xffffffff0,%esp
0x08048767 <nain+7>: pushl  -0x4(%ecx)
0x0804876a <nain+10>:      push    %ebp
0x0804876b <nain+11>:      mov     %esp,%ebp
0x0804876d <nain+13>:      push    %ecx
0x0804876e <nain+14>:      sub     $0x94,%esp
0x08048774 <nain+20>:      mov     %gs:0x14,%eax
0x0804877a <nain+26>:      mov     %eax,-0xc(%ebp)
0x0804877d <nain+29>:      xor     %eax,%eax
0x0804877f <nain+31>:      sub     $0x8,%esp
0x08048782 <nain+34>:      push    $0x0
0x08048784 <nain+36>:      lea     -0x80(%ebp),%eax
0x08048787 <nain+39>:      push    %eax
0x08048788 <nain+40>:      call    0x8048480 <gettimeofday@plt>
0x0804878d <nain+45>:      add     $0x10,%esp
0x08048790 <nain+48>:      sub     $0xc,%esp
0x08048793 <nain+51>:      push    $0x80489b4
0x08048798 <nain+56>:      call    0x8048460 <printf@plt>
0x0804879d <nain+61>:      add     $0x10,%esp
0x080487a0 <nain+64>:      mov     0x804a060,%eax
0x080487a5 <nain+69>:      sub     $0x4,%esp
0x080487a8 <nain+72>:      push    %eax
0x080487a9 <nain+73>:      push    $0x64
0x080487ab <nain+75>:      lea     -0x70(%ebp),%eax
0x080487ae <nain+78>:      push    %eax
0x080487af <nain+79>:      call    0x8048470 <fgets@plt>
0x080487b4 <nain+84>:      add     $0x10,%esp
0x080487b7 <nain+87>:      sub     $0xc,%esp
0x080487ba <nain+90>:      lea     -0x70(%ebp),%eax
0x080487bd <nain+93>:      push    %eax
0x080487be <nain+94>:      call    0x8048714 <transformar>
0x080487c3 <nain+99>:      add     $0x10,%esp
0x080487c6 <nain+102>:     nov     %eax,-0x88(%ebp)
0x080487cc <nain+108>:     sub     $0xc,%esp
0x080487cf <nain+111>:     push    $0x804a03c
0x080487d4 <nain+116>:     call    0x80484c0 <strlen@plt>
0x080487d9 <nain+121>:     add     $0x10,%esp
0x080487dc <nain+124>:     sub     $0x4,%esp
0x080487df <nain+127>:     push    %eax
0x080487e0 <nain+128>:     push    $0x804a03c
0x080487e5 <nain+133>:     pushl   -0x88(%ebp)
0x080487eb <nain+139>:     call    0x80484f0 <strncmp@plt>
0x080487f0 <nain+144>:     add     $0x10,%esp
0x080487f3 <nain+147>:     test    %eax,%eax
0x080487f5 <nain+149>:     je      0x80487fc <nain+156>
0x080487f7 <nain+151>:     call    0x804860b <boom>
```

En un primer vistazo podemos ver como en la posición **0x080487eb** se hace la llamada a **strncmp**, el cual es el encargado de ver si las dos cadenas son iguales. Si miramos el manual de **strncmp** sabemos que el primer parámetro es una cadena, el segundo una cadena y el tercero la longitud.

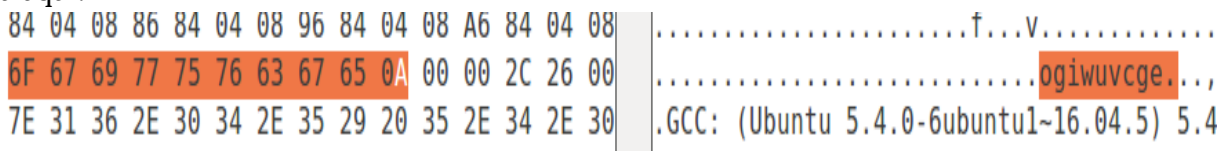
Por lo tanto podemos saber que el primer push que se realiza es el de la longitud de la cadena que estamos buscando. Para saber cual es vemos los registros y obtenemos que la longitud es **10**.



Ahora tenemos que averiguar donde se encuentra para saber a partir de donde tenemos que buscar. Podemos intuir que es el siguiente push. Pero para asegurarnos vamos a mirar donde guarda la llamada a “**transformar**” la encriptacion de la cadena que nosotros le hemos metido. Y como podemos ver en el pantallazo del disassembled, se guarda en `-0x88(%ebp)`, ya que al terminar la llamada realiza `mov %eax,-0x88(%ebp)`. Por lo tanto nos aseguramos de que el password se encuentra a partir de `$0x804a03c`. Para ver los 10 caracteres por los que esta compuesto el password usamos Data → Memory. Y buscamos 10 bytes a partir de esa posición. Cogemos el resultado en hexadecimal para luego buscarlo mas fácil en GHex.



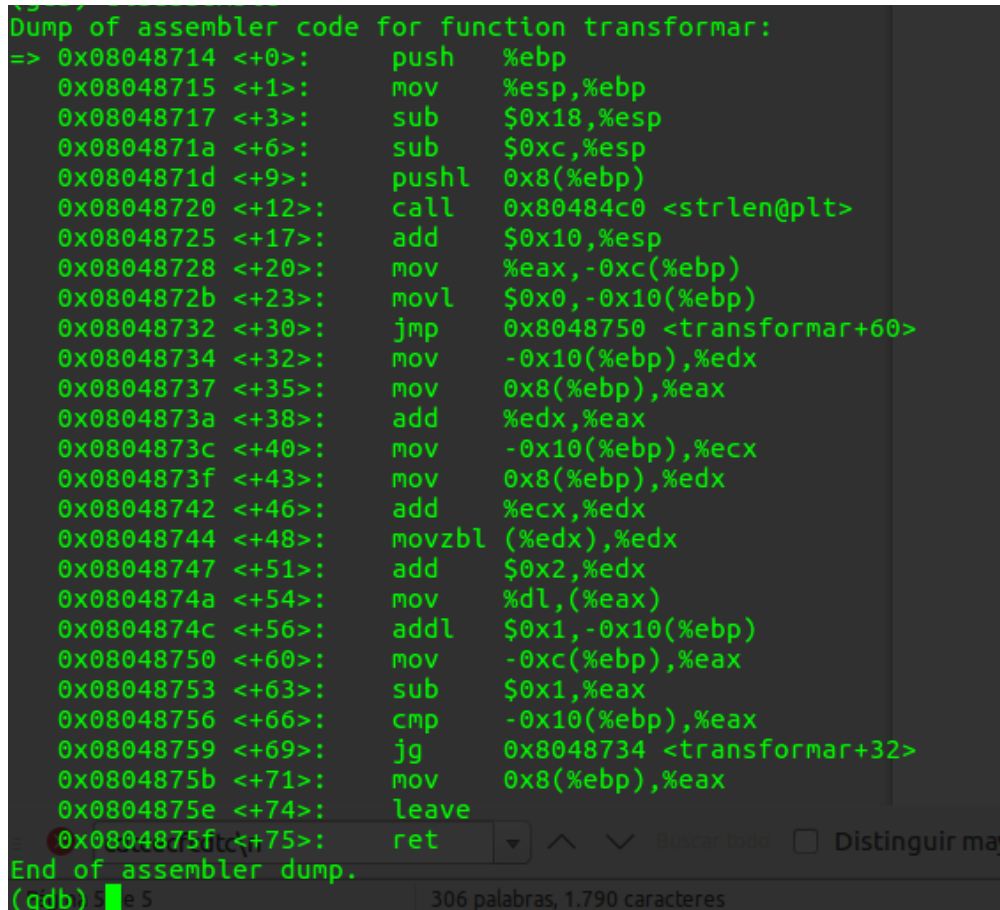
Ahora buscamos la secuencia 6f 67 690a con GHex. La secuencia ya la hemos encontrado y esta aquí:



Tenemos que el contenido de password es: **ogiwuvcge\n**

1.2 Método de encriptacion de clave alfanumérica.

Para ello vamos a ir directamente a la llamada a transformar. He tenido que usar gdb porque ddd me daba problemas a la hora de entrar en el call.



```
Dump of assembler code for function transformar:
=> 0x08048714 <+0>:      push    %ebp
      0x08048715 <+1>:      mov     %esp,%ebp
      0x08048717 <+3>:      sub     $0x18,%esp
      0x0804871a <+6>:      sub     $0xc,%esp
      0x0804871d <+9>:      pushl   0x8(%ebp)
      0x08048720 <+12>:     call    0x80484c0 <strlen@plt>
      0x08048725 <+17>:     add     $0x10,%esp
      0x08048728 <+20>:     mov     %eax,-0xc(%ebp)
      0x0804872b <+23>:     movl    $0x0,-0x10(%ebp)
      0x08048732 <+30>:     jmp     0x8048750 <transformar+60>
      0x08048734 <+32>:     mov     -0x10(%ebp),%edx
      0x08048737 <+35>:     mov     0x8(%ebp),%eax
      0x0804873a <+38>:     add     %edx,%eax
      0x0804873c <+40>:     mov     -0x10(%ebp),%ecx
      0x0804873f <+43>:     mov     0x8(%ebp),%edx
      0x08048742 <+46>:     add     %ecx,%edx
      0x08048744 <+48>:     movzbl  (%edx),%edx
      0x08048747 <+51>:     add     $0x2,%edx
      0x0804874a <+54>:     mov     %dl,(%eax)
      0x0804874c <+56>:     addl    $0x1,-0x10(%ebp)
      0x08048750 <+60>:     mov     -0xc(%ebp),%eax
      0x08048753 <+63>:     sub     $0x1,%eax
      0x08048756 <+66>:     cmp     -0x10(%ebp),%eax
      0x08048759 <+69>:     jg      0x8048734 <transformar+32>
      0x0804875b <+71>:     mov     0x8(%ebp),%eax
      0x0804875e <+74>:     leave
      0x0804875f <+75>:     ret
End of assembler dump.
(gdb) 5 306 palabras, 1,790 caracteres
```

A simple vista podemos ver que en **0x08048756** hace una comparación y luego salta transformar+32. Por lo tanto ahí podemos observar que se está realizando un bucle.

El resultado de lo que se está haciendo en el bucle se guarda de forma provisional en 0x08(%ebp) ya que al final lo mueve a %eax. Y el contador del for se puede observar que está en **-0x10(%ebp)** ya que el lo que compara con %eax antes de hacer el **jg**. Por lo tanto sabiendo esto podemos ver como desde +40 a +46 está moviendo la posición del char[] la cual estamos modificando y lo está guardando en %edx. Una vez que sabemos donde está podemos ver que está modificando cada carácter sumándole **0x2** en +51.

Por lo tanto sabemos que está realizando un algoritmo que está sumándole dos a el valor en int de todos los caracteres del password.

Como teníamos que la clave encriptada es **ogiwuvcge**, le restamos 2 a cada una de sus letras por separado y tenemos que la password sin encriptar es **megustaec**.

2 Clave numérica

2.1 Buscando la clave numérica encriptada

Nos vamos a buscar en la siguiente parte del código:

```
0x08048821 <+193>: sub    $0xc,%esp
0x08048824 <+196>: push   $0x80489cf
0x08048829 <+201>: call   0x8048460 <printf@plt>
0x0804882e <+206>: add    $0x10,%esp
0x08048831 <+209>: sub    $0x8,%esp
0x08048834 <+212>: lea    -0x8c(%ebp),%eax
0x0804883a <+218>: push   %eax
0x0804883b <+219>: push   $0x80489e6
0x08048840 <+224>: call   0x80484e0 <__isoc99_scanf@plt>
0x08048845 <+229>: add    $0x10,%esp
0x08048848 <+232>: mov    -0x8c(%ebp),%eax
0x0804884e <+238>: sub    $0xc,%esp
0x08048851 <+241>: push   %eax
0x08048852 <+242>: call   0x80486c4 <encriptar>
0x08048857 <+247>: add    $0x10,%esp
0x0804885a <+250>: mov    %eax,-0x84(%ebp)
0x08048860 <+256>: mov    0x804a04c,%eax
0x08048865 <+261>: cmp    %eax,-0x84(%ebp)
0x0804886b <+267>: je     0x8048872 <main+274>
0x0804886d <+269>: call   0x804860b <boom>
0x08048872 <+274>: sub    $0x8,%esp
0x08048875 <+277>: push   $0x0
0x08048877 <+279>: lea    -0x80(%ebp),%eax
0x0804887a <+282>: push   %eax
0x0804887b <+283>: call   0x8048480 <gettimeofday@plt>
0x08048880 <+288>: add    $0x10,%esp
0x08048883 <+291>: mov    -0x80(%ebp),%edx
0x08048886 <+294>: mov    -0x78(%ebp),%eax
0x08048889 <+297>: sub    %eax,%edx
0x0804888b <+299>: mov    %edx,%eax
0x0804888d <+301>: cmp    $0x5,%eax
0x08048890 <+304>: jle    0x8048897 <main+311>
0x08048892 <+306>: call   0x804860b <boom>
0x08048897 <+311>: call   0x804864b <defused>
0x0804889c <+316>: mov    $0x0,%eax
0x080488a1 <+321>: mov    -0xc(%ebp),%ecx
0x080488a4 <+324>: xor    %gs:0x14,%ecx
0x080488ab <+331>: je     0x80488b2 <main+338>
0x080488ad <+333>: call   0x8048490 <__stack_chk_fail@plt>
0x080488b2 <+338>: mov    -0x4(%ebp),%ecx
0x080488b5 <+341>: leave  %ecx
0x080488b6 <+342>: lea    -0x4(%ecx),%esp
0x080488b9 <+345>: ret
```

Podemos ver como en +261 esta realizando la comparación para ver si es correcta o no. Por lo tanto la clave encriptada debe esta en %eax o en -0x84(%ebp). Sabemos que en -0x84(%ebp) va a estar lo que nosotros hayamos introducido encriptado ya que justo al salir de la llamada a encriptar

mueve **%eax** ahí. Siendo %eax el registro donde devuelve encriptar el resultado, ya que es esa la convección, aunque también lo podríamos ver viendo el disassembled de la llamada a encriptar.

Por lo tanto para saber cual es el passcode que guarda encriptado el código solo tenemos que hacer el break en **+261** y ver la información de los registros.

```
Breakpoint 1, 0x0804876e in main ()
(gdb) continue
Continuando.
Introduce la contraseña: abracadabra
Introduce el código: 7777

Breakpoint 2, 0x08048865 in main ()
(gdb) info register
eax             0x262c      9772
ecx             0x7        7
edx             0x1e61     7777
ebx             0x0        0
esp             0xffffcd30   0xffffcd30
ebp             0xffffcdc8   0xffffcdc8
esi             0xf7fa9000   -134574080
edi             0xf7fa9000   -134574080
eip             0x8048865    0x8048865 <main+261>
```

Viendo esto sabemos que el passcode original encriptado es **9772**. Y en edx se encuentra lo que a introducido el usuario.

2.2 Método de encriptacion de clave numérica.

Nos vamos directamente al código de la llamada a encriptar.

```
Dump of assembler code for function encriptar:
=> 0x080486c4 <+0>:      push    %ebp
0x080486c5 <+1>:      mov     %esp,%ebp
0x080486c7 <+3>:      sub     $0x10,%esp
0x080486ca <+6>:      pushl   0x8(%ebp)
0x080486cd <+9>:      call    0x804868b <longitud>
0x080486d2 <+14>:     add     $0x4,%esp
0x080486d5 <+17>:     mov     %eax,-0x4(%ebp)
0x080486d8 <+20>:     movl    $0x7,-0xc(%ebp)
0x080486df <+27>:     movl    $0x0,-0x8(%ebp)
0x080486e6 <+34>:     jmp     0x80486ff <encriptar+59>
0x080486e8 <+36>:     mov     -0xc(%ebp),%edx
0x080486eb <+39>:     mov     %edx,%eax
0x080486ed <+41>:     shl     $0x2,%eax
0x080486f0 <+44>:     add     %edx,%eax
0x080486f2 <+46>:     add     %eax,%eax
0x080486f4 <+48>:     mov     %eax,-0xc(%ebp)
0x080486f7 <+51>:     addl    $0x7,-0xc(%ebp)
0x080486fb <+55>:     addl    $0x1,-0x8(%ebp)
0x080486ff <+59>:     mov     -0x4(%ebp),%eax
0x08048702 <+62>:     sub     $0x1,%eax
0x08048705 <+65>:     cmp     -0x8(%ebp),%eax
0x08048708 <+68>:     jg      0x80486e8 <encriptar+36>
0x0804870a <+70>:     mov     0x8(%ebp),%edx
0x0804870d <+73>:     mov     -0xc(%ebp),%eax
0x08048710 <+76>:     add     %edx,%eax
0x08048712 <+78>:     leave
0x08048713 <+79>:     ret
End of assembler dump.
(qdb) 6 669 palabras, 3.799 caracteres
```

Aquí como podemos ver al igual que en el método de la clave alfanumérica podemos ver que hay también un bucle que va desde +36 a +68. En un simple vistazo podemos ver que para el bucle en -0x8(%ebp) guarda el índice y que en %eax se encuentra el número de vueltas que va a dar el bucle. En %eax está guardado el valor de retorno de la llamada a longitud. Que a priori no sabemos que es, aunque podemos atisbar que va a ser la cantidad de dígitos del número pasado por el parámetro. Para ello podemos ver el código de la llamada o para hacerlo más rápido meter un código y cuando llegue a “encriptar+14” ver el contenido de %eax y ver si corresponde. Sabemos que si es así. Por lo tanto pasamos a ver cuál es el método de encriptación.

En %edx se está conformando el valor que se le suma al passcode para darlo encriptado, ya que en +76 realiza la suma de edx y eax. Ahora tenemos que ver cuál es el valor que genera para sumárselo al passcode introducido por nosotros. Viendo que el código ensamblador resultante es un poco engorroso para averiguar el algoritmo, podemos hacer algo más fácil y es irnos a +76 y ver cuál es el contenido que le suma a %eax. Y así es bastante más fácil que tener que entender todo ese código ensamblador que hay dentro del for.

Y obtenemos:

```
0x08048713 <+7777> 7777
End of assembler dump.
(gdb) info registers
eax            0x1e61    7777
ecx            0x1      1
edx            0x457    1111
ebx            0x0      0
esp            0x5555d00 0x5555d00
```

Por lo tanto podemos ver que esta sumando 7777 y como al ver lo que nos devolvía longitud vimos que devolvía 4, podemos imaginar que lo que realiza es sumar 7 a cada uno de los dígitos del passcode introducido.

Como obtuvimos anteriormente que el passcode encriptado era 9772 le restamos 7777 y obtenemos que el passcode sin encriptar es 1995.

Lo probamos y vemos que tanto password y passcode son correctos.

```
antonio-Lenovo ~ /Documents/git/PRACTICAS-EC/Practica3:
3551 ± ./bomba_mia
Introduce la contraseña: megustaec
Introduce el código: 1995
.....
... bomba desactivada ...
.....
```