QBIT ACADEMY // MODULE 01

SQL INJECTION I: FUNDAMENTOS

CICLO TÉCNICO 2026 — INSTRUCTOR SENIOR

01. ARQUITECTURA DE LA CONSULTA

EL MOTOR SQL

Para entender SQLi, primero debemos entender cómo una base de datos procesa la información:

- **Parser:** Analiza la sintaxis y verifica errores gramaticales.
- **Optimizer:** Crea el plan de ejecución más eficiente.
- **Executor:** Recupera los datos del almacenamiento físico.

La vulnerabilidad nace en el Parser.

Si el atacante inyecta código, el Parser lo interpreta como **instrucciones** y no como **datos**, alterando el flujo lógico.

02. PLANO DE CONTROL VS DATOS

PLANO DE CONTROL

Son las palabras reservadas (SELECT, FROM, WHERE) que definen la estructura de la consulta.

PLANO DE DATOS

Es la información variable suministrada por el usuario (ej: ID, Username, Email).

Fallo Técnico: La concatenación de strings mezcla ambos planos. El motor no puede distinguir dónde termina el comando y dónde empieza el dato.

03. CONCATENACIÓN EN EL BACKEND

Observemos cómo el código construye la sentencia internamente:

```
// Código en PHP/NodeJS/Python
$id = $_GET['id'];
$query = "SELECT * FROM products WHERE id = '" . $id . "'";
```

Si el usuario envía 10:

`... WHERE id = '10'` (Válido)

Si el usuario envía 10':

`... WHERE id = '10''` (Syntax Error)

La comilla simple inyectada cierra el Plano de Datos prematuramente y permite al atacante escribir en el Plano de Control.

04. ANATOMÍA DE LA INYECCIÓN

Cuando el Parser recibe la cadena, realiza una **Tokenización**. Una sola comilla altera los tokens generados:

```
TOKEN_CMD: SELECT
 TOKEN_FROM: products
 TOKEN_WHERE: id
 TOKEN_DATA: 10 ⟵ Cierre inesperado
TOKEN_UNKNOWN: '
```

BREAK & FIX

Este es el "Oráculo del Error":

Break: Inyectamos ' para romper la integridad.

Fix: Inyectamos lógica para reparar la query y confirmar control.

05. ESTRATEGIA DE IDENTIFICACIÓN

ERROR-BASED

Buscamos mensajes de error detallados (HTTP 500) que revelen información del motor.

BOOLEAN-BASED

Inferimos la vulnerabilidad enviando condiciones TRUE y FALSE y observando cambios en el HTTP 200.

06. LAB 1: HIDDEN DATA

Escenario: Filtro de productos. Solo deberían verse los productos donde released = 1.

```
-- Query Esperada:
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

ATAQUE:

Input: Gifts' OR 1=1--

La query resultante anula el filtro comercial:

```
WHERE category = 'Gifts' OR 1=1 --' AND
released = 1
```

07. LA LÓGICA DE LA TAUTOLOGÍA

En lógica matemática, una **tautología** es una fórmula que siempre es verdadera independientemente de los valores.

Condición A	Operador	Condición B	Resultado
FALSE	OR	TRUE (1=1)	**TRUE**
TRUE	OR	TRUE (1=1)	**TRUE**

Al forzar un OR TRUE en el WHERE, la base de datos devuelve **todas** las filas de la tabla porque la condición se cumple para cada registro.

08. COMENTARIOS POR SINTAXIS

Base de Datos	Sintaxis de Comentario	Ejemplo de Payload
MySQL / MariaDB	# o -- (espacio)	' OR 1=1 #
PostgreSQL	--	' OR 1=1 --
MSSQL (SQL Server)	--	' OR 1=1 --
Oracle	--	' OR 1=1 --

*Nota: En MySQL, el espacio después de los guiones es obligatorio para que se reconozca como comentario.

09. TEORÍA DE VALIDACIÓN DE LOGIN

¿Cómo sabe una web que la contraseña es correcta?

```
SELECT * FROM users WHERE
user = '?' AND pass = '?'
```

Si la DB devuelve **num_rows > 0**, la aplicación crea la sesión.

EL OBJETIVO DEL ATAQUE

No necesitamos saber la contraseña. Solo necesitamos que la consulta devuelva **al menos un registro** con el nombre del usuario deseado.

10. LAB 2: LOGIN BYPASS

ESCENARIO REAL:

Campo Usuario: administrator'--

Campo Contraseña: (Cualquiera)

```
-- Query Final Ejecutada:
SELECT * FROM users WHERE user = 'administrator'--' AND pass = '...'
```

El motor SQL busca al usuario 'administrator', lo encuentra, y el resto (la comprobación del password) es **ignorado** por el comentario. Acceso concedido.

11. EL OPERADOR UNION

El comando UNION permite combinar el resultado de dos consultas distintas en un solo set de datos.

REGLA 1: COLUMNAS

Ambas consultas deben tener el **mismo número** de columnas.

REGLA 2: TIPOS

Los tipos de datos deben ser **compatibles** (Afinidad de datos).

Si la query original devuelve productos, y nuestra inyección pide usuarios, los datos de los usuarios aparecerán **anexados** en la interfaz de la tienda.

12. LAB 3: ENUMERACIÓN DE COLUMNAS

MÉTODO ORDER BY

Es más rápido que UNION para contar columnas:

```
' ORDER BY 1-- (OK)
' ORDER BY 2-- (OK)
' ORDER BY 3-- (ERROR)
```

MÉTODO UNION NULL

Usamos NULL porque es compatible con cualquier tipo:

```
' UNION SELECT NULL--
' UNION SELECT NULL,NULL--
```

13. DEFENSA: PREPARED STATEMENTS

Inseguro (Concatenación)

Mezcla datos y comandos en el mismo canal.

Seguro (Parametrización)

Envía la estructura y los datos por canales separados.

¿Por qué funciona? Al usar Prepared Statements, el motor SQL **pre-compila** la query. Cuando llega el dato del usuario, ya es tarde para que el Parser lo interprete como código; se queda siempre en el Plano de Datos.

// INTRO: Diferenciar de SQLi. Aquí no inyectamos strings, inyectamos objetos.

NOSQL INJECTION

MÁS ALLÁ DE LAS TABLAS //

Explotación de Bases de Datos Orientadas a Documentos

// EXPLICAR: Por qué las empresas usan NoSQL (escalabilidad) y por qué los desarrolladores se confían.

01. EL ECOSISTEMA NOSQL

A diferencia de las DB relacionales, NoSQL (como **MongoDB**) almacena datos en documentos flexibles, usualmente en formato **BSON/JSON**.

- **Colecciones:** Equivalentes a Tablas.
- **Documentos:** Equivalentes a Filas.
- **Sin Esquema:** No requiere estructura fija.

EJEMPLO DE DOCUMENTO

```
{
  "_id": "507f1f",
  "user": "admin",
  "role": "superadmin",
  "active": true
}
```

// EXPLICAR: En SQLi atacamos al Parser de Strings. En NoSQLi atacamos al Query Engine de Objetos.

02. CAMBIO DE PARADIGMA

SQL INJECTION
Manipula la **cadena de texto** de la query antes de ser parseada.

```
' OR 1=1 --
```

NOSQL INJECTION
Manipula la **estructura del objeto** de consulta (operadores).

```
{"$ne": ""}
```

El ataque ocurre cuando la aplicación acepta **objetos** del usuario en lugar de solo strings, permitiendo cambiar la lógica de la búsqueda.

// EXPLICAR: Estos operadores son nativos. El motor los ejecuta si los recibe en el lugar correcto.

03. EL DICCIONARIO DEL ATACANTE

En MongoDB, las consultas usan operadores especiales. Si podemos inyectar estos operadores, alteramos el resultado.

Operador	Significado	Uso en Ataque
$eq	Igual a (Equal)	Consulta estándar.
$ne	No es igual (Not Equal)	**Bypass de Login**: busca algo que NO sea vacío.
$gt / $lt	Mayor / Menor que	Enumeración de datos numéricos.
$in	Dentro de un array	Probar múltiples valores simultáneamente.

// EXPLICAR: El peligro de pasar req.body directamente a la DB sin tipar.

04. ¿CÓMO OCURRE LA INYECCIÓN?

En entornos como **Node.js (Express)** o **PHP**, si no se valida el tipo de dato, un atacante puede enviar un objeto JSON a través de una petición HTTP.

```
// Backend vulnerable
db.users.find({
  user: req.body.user,
  pass: req.body.pass
})
```

INPUT MALICIOSO

Si el atacante envía un JSON en el body:

```
{
  "user": "admin",
  "pass": {"$ne": "1"}
}
```

La consulta busca un usuario admin cuya password **no sea "1"**.

// EXPLICAR: Cómo usar Burp Suite para cambiar un parámetro "user=admin" por "user[$ne]=1".

05. METODOLOGÍA DE DETECCIÓN

1. INYECCIÓN DE TIPOS

Cambiar parámetros de string a array u objeto en la petición HTTP.

```
?user[$ne]=null
```

Observamos la respuesta: si el servidor devuelve un **Error 500** (indica fallo de lógica) o un **Bypass exitoso**, la entrada no está sanitizada.

// EXPLICAR: Por qué devuelve el primer registro. MongoDB .find() devuelve un cursor o .findOne().

06. LAB 1: LOGIN BYPASS CON $NE

OBJETIVO: ENTRAR COMO CUALQUIER USUARIO

Usamos el operador **"Not Equal"** para que la base de datos devuelva cualquier documento que cumpla la condición de no ser igual a un valor absurdo.

```
Petición POST (Content-Type: application/json)
{
    "username": {"$ne": null},
    "password": {"$ne": null}
}
```

La DB responderá: "Dame el primer usuario donde el nombre no sea null y el pass no sea null". **Acceso concedido al primer registro (usualmente el admin).**

// EXPLICAR: Diferencia entre enviar el string "$ne" y enviar el objeto {"$ne": ""}.

07. POR QUÉ FUNCIONA EL $NE

LÓGICA TRADICIONAL

admin == admin (TRUE)

pass == secret (TRUE)

Requiere conocer ambos.

LÓGICA INYECTADA

admin == admin (TRUE)

"123" != "" (TRUE)

La condición se cumple para cualquier password existente.

El motor NoSQL evalúa la expresión del objeto. Si el objeto {"$ne": ""} es procesado, la comparación de igualdad se convierte en una comparación de desigualdad.

// EXPLICAR: Esta es una técnica Blind (Ciega). Requiere automatización.

08. LAB 2: EXFILTRACIÓN CON $REGEX

Cuando no podemos saltar el login directamente, podemos **extraer datos** carácter a carácter usando expresiones regulares.

EL OPERADOR $REGEX

Permite buscar patrones. Podemos preguntar si la contraseña empieza por 'a', luego por 'ab', etc.

```
{"pass": {"$regex": "^a"}}
```

Si la respuesta es "Login Fallido", probamos con 'b'.
Si la respuesta es "Login Exitoso", sabemos que empieza con 'a' y pasamos al siguiente carácter.

// EXPLICAR: Cómo usar 'Intruder' de Burp o un script en Python para automatizar esto.

09. EL PROCESO DE ENUMERACIÓN

PASO 1

^a.* → 200 OK

Inicia con 'a'

PASO 2

^ad.* → 200 OK

Sigue 'd'

PASO 3

^adm.* → 200 OK

Sigue 'm'

PASO 4

^admin.* → 200 OK

¡Encontrado!

// EXPLICAR: Esto es equivalente a un 'eval()' en JavaScript. Extremadamente peligroso.

10. EL PELIGRO DEL OPERADOR $WHERE

Algunas funciones de MongoDB permiten ejecutar **JavaScript puro** en el servidor para filtrados complejos.

Si un atacante inyecta código JS en un operador $where, puede lograr **RCE (Remote Code Execution)** limitado o denegación de servicio.

EJEMPLO CRÍTICO

```
db.collection.find({
  $where: "this.user == '" + input + "'"
})
```

Input: '; while(true){}; var x='

// EXPLICAR: Muy útil en APIs que siempre devuelven {"status": "ok"}.

11. INFERENCIA POR TIEMPO

Si la aplicación no devuelve mensajes distintos para éxito o fallo, usamos **Time-based Injection** inyectando JavaScript.

```
{ "$where": "if (this.user == 'admin') sleep(5000)" }
```

RESPUESTA RÁPIDA
La condición es falsa. El usuario no es admin.

RESPUESTA LENTA (5S)
La condición es verdadera. ¡Hemos confirmado el nombre de usuario!

// EXPLICAR: Por qué NoSQLi es a veces más difícil de detectar con scanners automáticos.

12. TABLA COMPARATIVA FINAL

Característica	SQL Injection	NoSQL Injection
Vector	Strings / Comillas	Objetos / Operadores
Objetivo	Parser de SQL	Query Engine / BSON Wrapper
Detección	Errores de sintaxis	Errores de tipo / Lógica
Herramienta	sqlmap	nosqlmap / Scripts manuales

// EXPLICAR: La importancia de la "Defensa en Profundidad".

13. BLINDAJE DE LA BASE DE DATOS

1. CASTEO DE TIPOS

Asegurarse de que el input sea un string y no un objeto.

```
String(req.body.user)
```

2. SANITIZACIÓN

Usar librerías como mongo-sanitize que eliminan cualquier llave que empiece con $.

3. ESQUEMAS (MONGOOSE)

Utilizar ODMs con esquemas estrictos que rechacen automáticamente objetos donde se esperan valores planos.

// MOTIVACIÓN: Invitar a los alumnos a probar NoSQLmap en sus entornos locales.

¿DUDAS?

LA INYECCIÓN NO MUERE, SOLO EVOLUCIONA.