

SQL Injection I: Fundamentos de Explotación



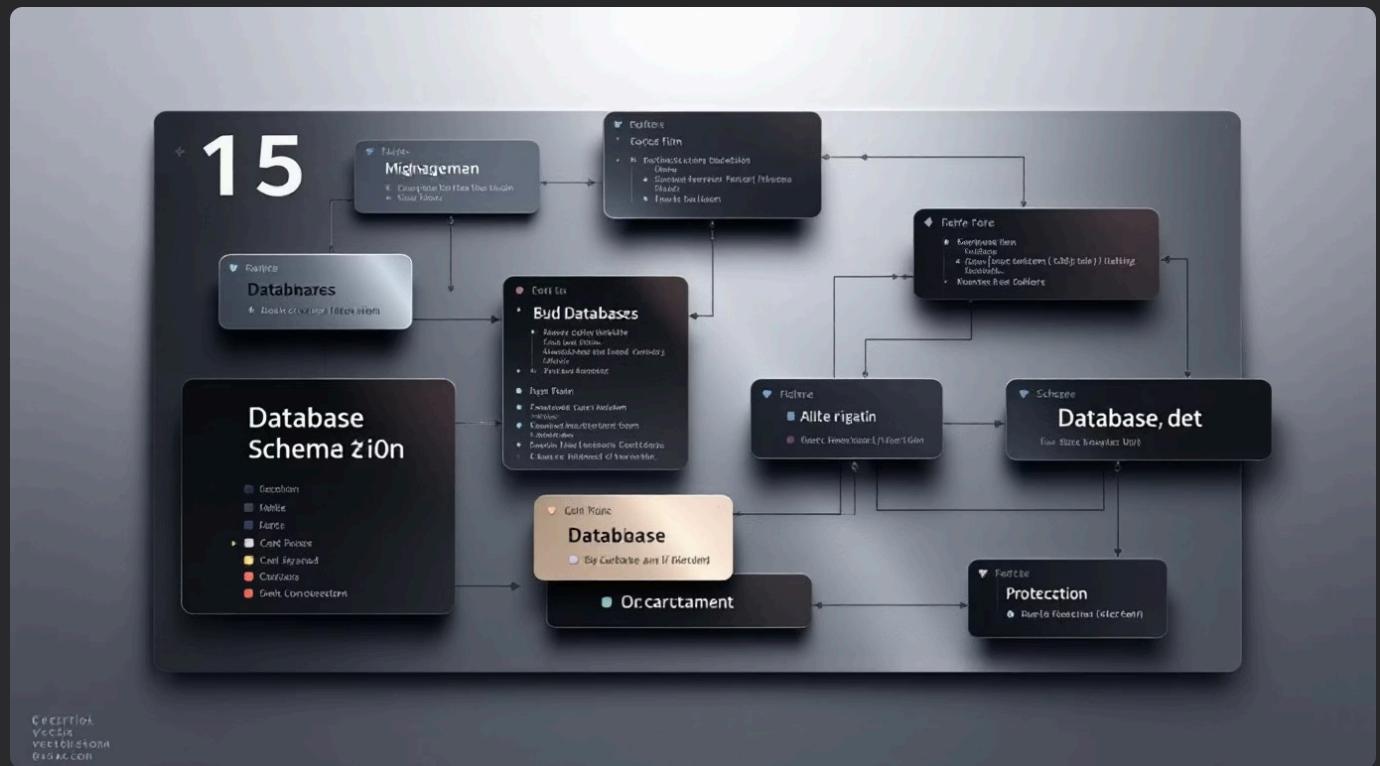
¿Qué es una Base de Datos Relacional?

El Modelo Relacional

Las bases de datos relacionales organizan la información en tablas estructuradas. Cada tabla contiene columnas (campos) y filas (registros), permitiendo almacenar y consultar datos de forma eficiente.

El Intérprete SQL

El motor de base de datos (MySQL, PostgreSQL, MSSQL) actúa como intérprete: recibe instrucciones SQL, las procesa y devuelve resultados. Este flujo es crítico para entender las inyecciones.



SELECT name FROM users WHERE id = 1;

Ejemplo básico: consulta que recupera el nombre del usuario con ID 1

La Raíz del Mal: Concatenación

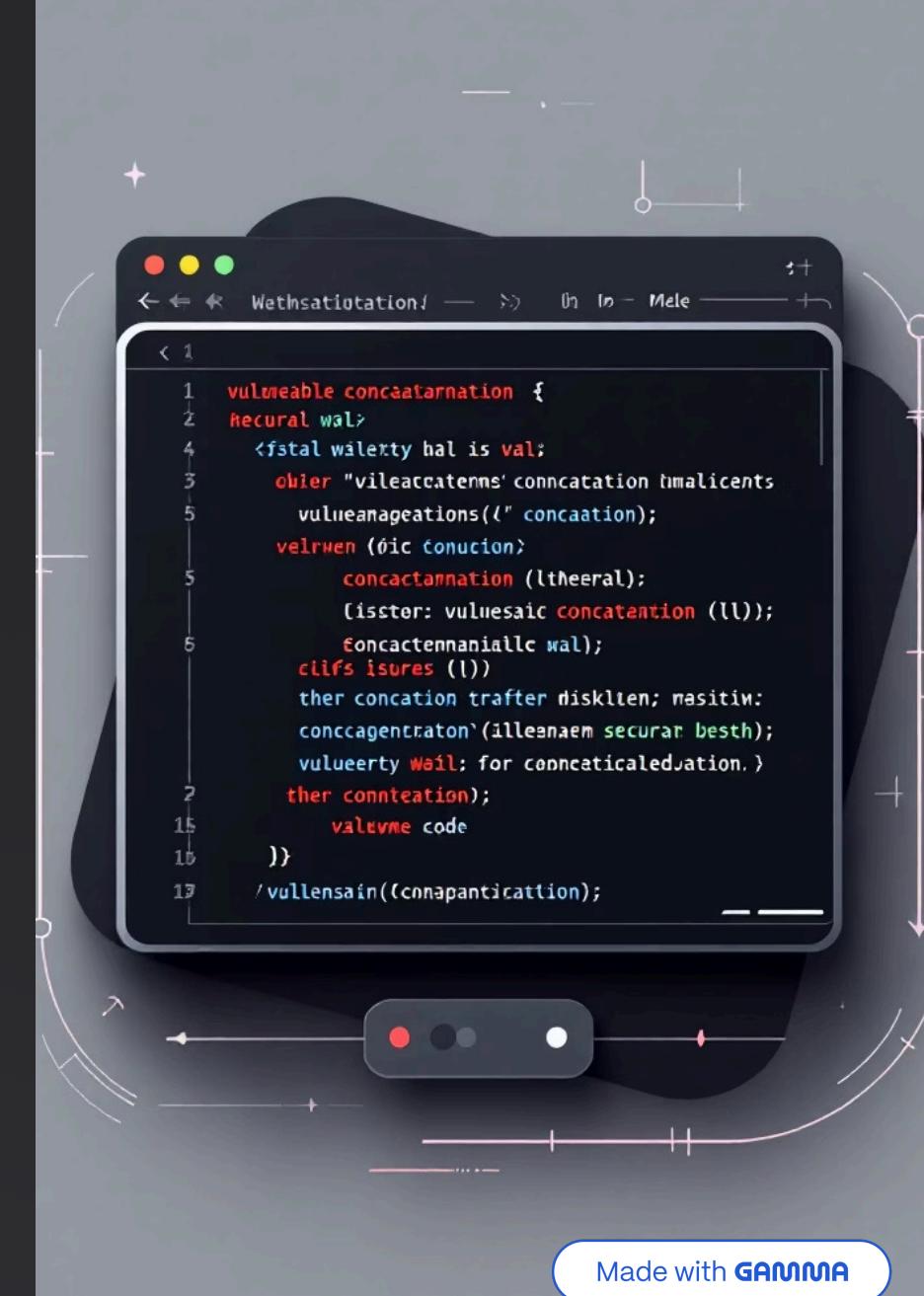
El Error Fundamental

El problema surge cuando mezclamos **código** (instrucciones SQL) con **datos** (entrada del usuario). Sin separación clara, el usuario puede alterar la lógica de la consulta.

El Fallo en Acción

```
$query = "SELECT * FROM users  
WHERE id = '" . $user_id . "'";
```

Esta concatenación directa permite que el atacante escriba parte del código SQL, transformando datos en instrucciones ejecutables.



Anatomía de la Comilla ('')



El Delimitador Crítico

La comilla simple es el símbolo que delimita cadenas de texto en SQL. Al injectar una comilla, cerramos prematuramente el campo de datos y abrimos el contexto de control.

Provocando el Error

```
... WHERE id = '1"
```

Este input genera un error de sintaxis que confirma vulnerabilidad: la aplicación no sanitiza la entrada y el motor SQL intenta procesar nuestra comilla.

Metodología del Auditor



1. Break

Provocar un error 500 o comportamiento inesperado para confirmar que nuestra entrada llega al motor de base de datos sin filtrado.



2. Fix

Reparar la consulta usando comentarios SQL (-- o #) para confirmar que controlamos la ejecución y validar la vulnerabilidad.

- Este flujo sistemático distingue un error aleatorio de una vulnerabilidad explotable de SQL Injection.

La Lógica del OR 1=1

Tautologías en SQL

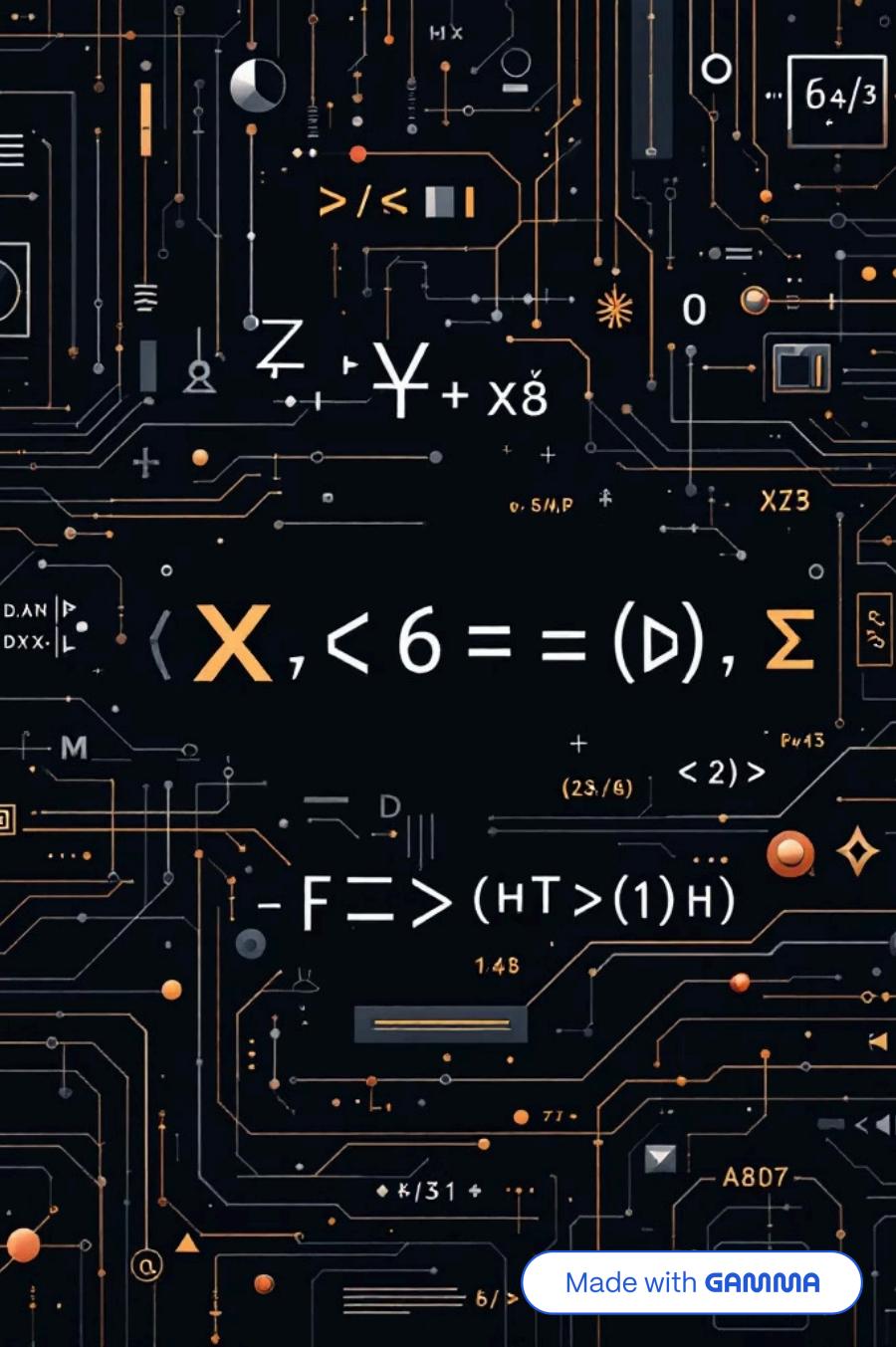
Una tautología es una condición siempre verdadera. En SQL, 1=1 siempre evalúa como TRUE, independientemente de otros factores.

Tabla de Verdad

TRUE OR FALSE =
TRUE
TRUE OR TRUE =
TRUE
Cualquier condición combinada con OR y una tautología siempre retorna TRUE.

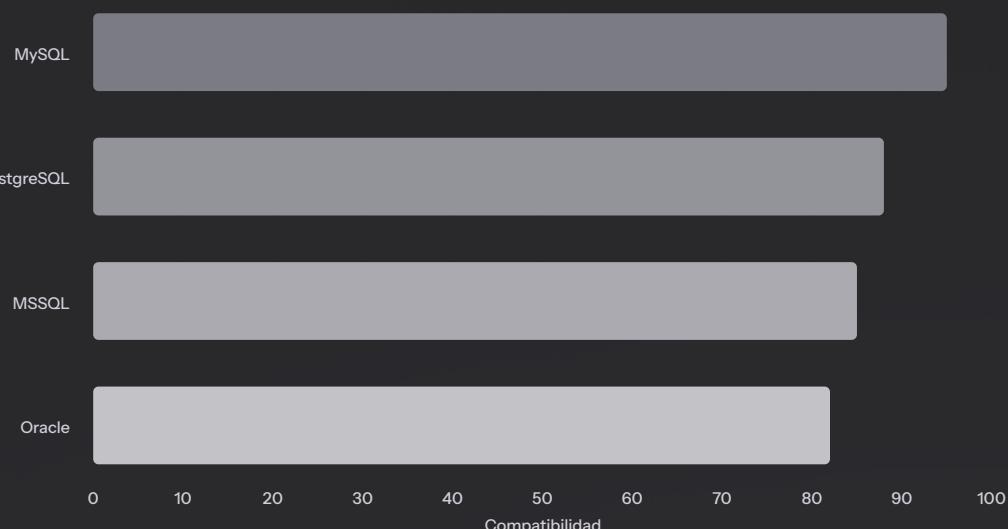
Impacto Devastador

Al inyectar OR 1=1, anulamos completamente la cláusula WHERE, forzando a la base de datos a devolver **todos los registros** de la tabla.



Comentarios según el Motor (Fingerprinting)

Motor SQL



Sintaxis por Motor

- **MySQL:** # o -- (con espacio obligatorio)
- **PostgreSQL:** --
- **MSSQL:** --
- **Oracle:** --

Los comentarios ignoran el resto de la query original, permitiendo eliminar validaciones posteriores a nuestra inyección.

Bypass de Autenticación

Consulta Original

```
SELECT * FROM users  
WHERE user='admin'  
AND pass='...'
```

Query Resultante

```
SELECT * FROM users  
WHERE user='admin'--
```

La validación de contraseña desaparece.

1

2

3

4

Inyección

Usuario: admin'--

Contraseña: (cualquiera)

Resultado

Acceso concedido sin credenciales válidas. El motor encuentra al admin y el comentario elimina la verificación.

El Operador UNION



Combinación de Consultas

UNION permite combinar resultados de dos queries distintas en un único conjunto de datos. Es la técnica fundamental para extraer información de tablas no previstas.

Reglas de Oro

- Número de columnas:** Ambas queries deben retornar el mismo número de columnas
- Tipos compatibles:** Los tipos de datos deben coincidir en posición



```
SELECT name, email FROM users WHERE id=1  
UNION  
SELECT username, password FROM admin
```

Prevención: Prepared Statements

La Solución Definitiva

Los prepared statements separan completamente código de datos. El motor compila la query **antes** de recibir los parámetros del usuario.

Mecánica de Protección

Como la estructura SQL está precompilada, los datos de entrada nunca pueden ser interpretados como instrucciones ejecutables, eliminando el vector de ataque.

X Código Inseguro

```
$query = "SELECT * FROM  
users  
WHERE id = '" . $id . "'";
```

✓ Código Seguro

```
$stmt = $pdo->prepare(  
    "SELECT * FROM users  
    WHERE id = :id"  
);  
$stmt->execute(['id' => $id]);
```

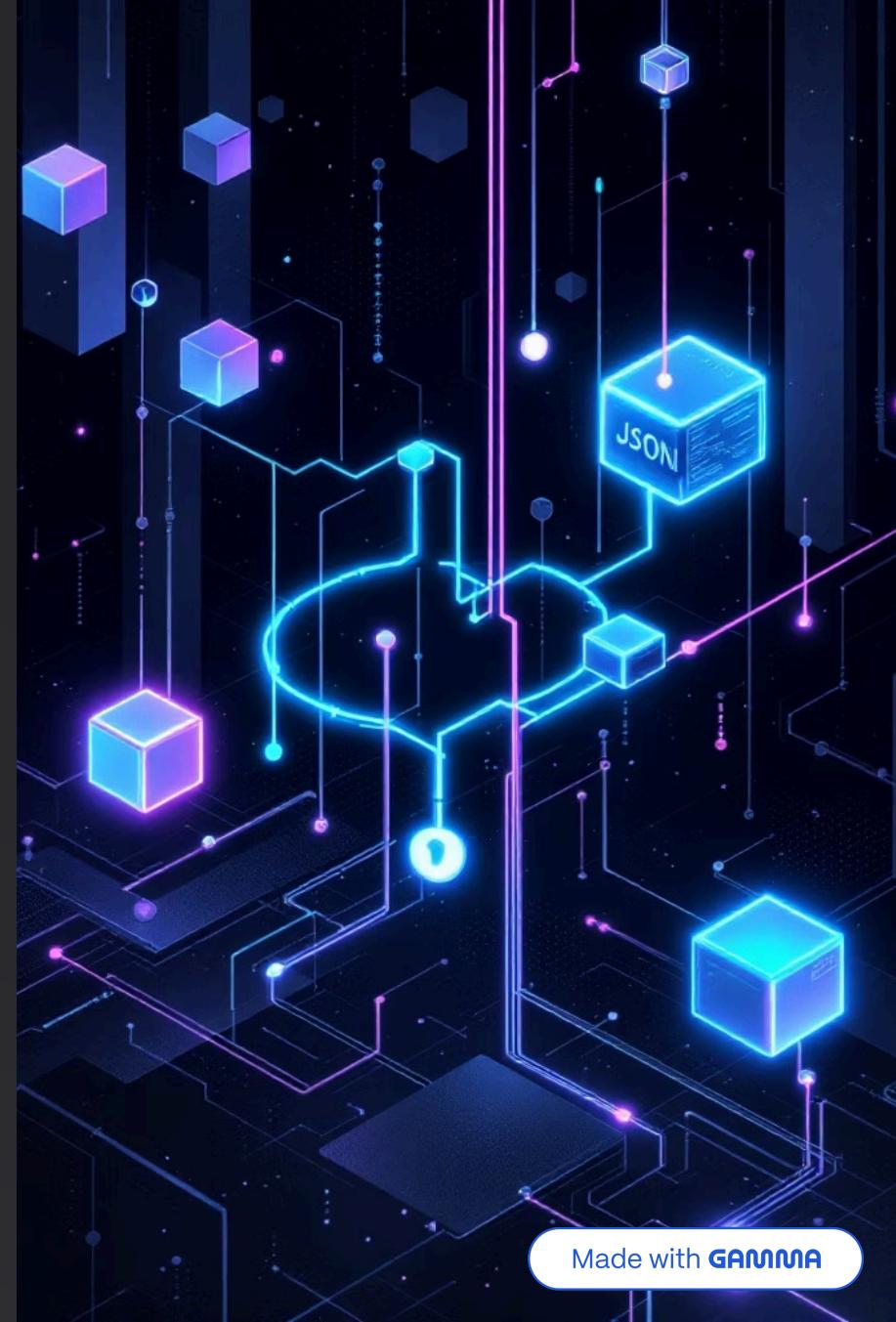


NoSQL Injection: Hacking Documental

QBIT SYSTEMS ACADEMY

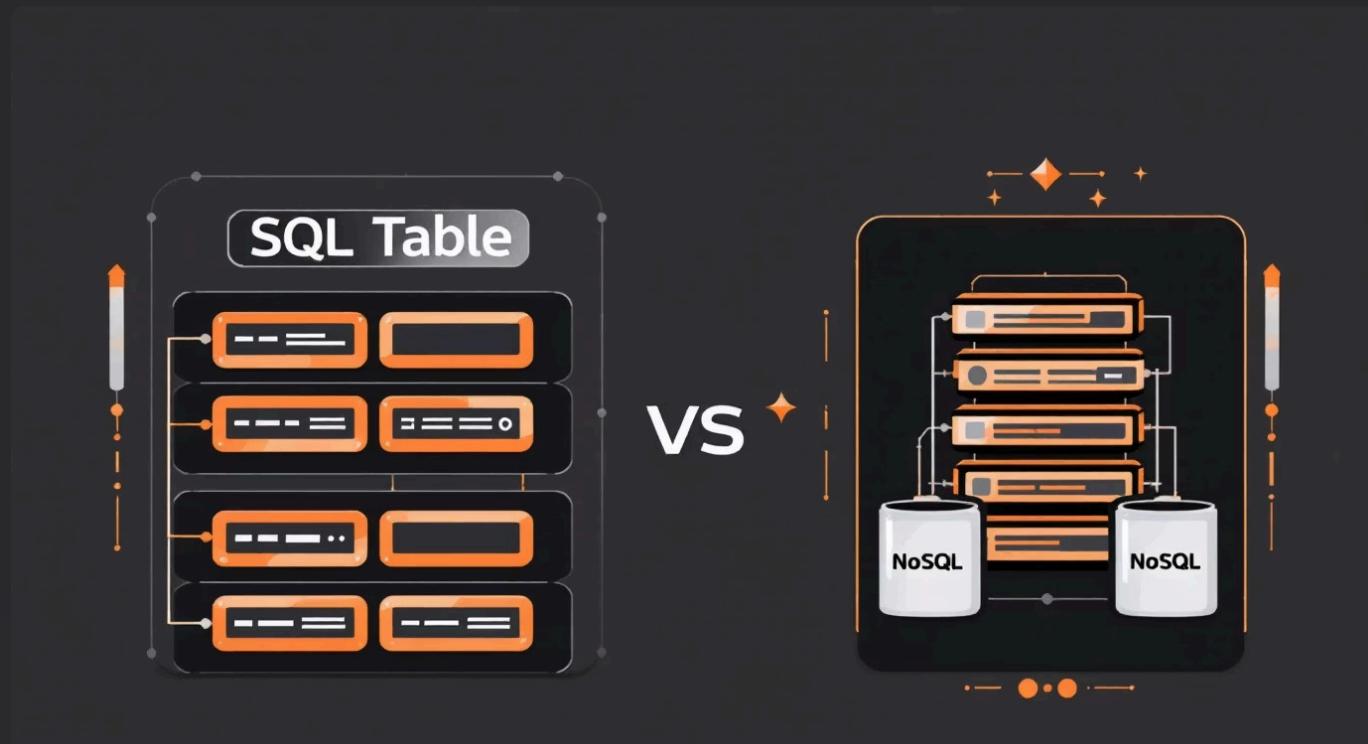
CICLO 2026

Adentrándose en las vulnerabilidades de las bases de datos no relacionales



Made with GAMMA

El Mundo Sin Tablas: MongoDB y NoSQL

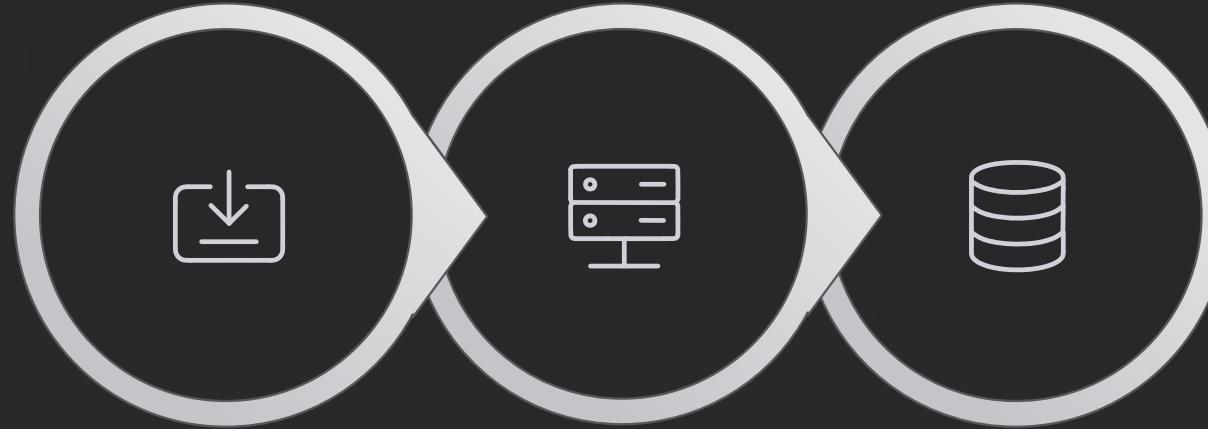


Cambio de Paradigma

A diferencia de SQL, que organiza datos en tablas y filas, NoSQL utiliza documentos y colecciones almacenados en formato BSON/JSON. Esta arquitectura flexible elimina las relaciones rígidas pero introduce nuevos vectores de ataque.

```
{  
  "user": "admin",  
  "active": true,  
  "role": "administrator"  
}
```

El Cambio de Paradigma: Inyección de Objetos



Input JSON

Procesa
objeto

Operadores
maliciosos

En NoSQL no inyectamos cadenas de texto como en SQL. El verdadero peligro surge cuando inyectamos **objetos JSON completos con operadores nativos**. La vulnerabilidad ocurre cuando el backend acepta un objeto directamente en la función de consulta sin validación ni sanitización previa.

Operadores de Comparación en MongoDB

\$ne (Not Equal)

Selecciona documentos donde el campo no es igual al valor especificado. Equivalente al operador != en SQL.

```
{"edad": {"$ne": 18}}
```

\$gt (Greater Than)

Selecciona documentos donde el campo es mayor que el valor especificado. Equivalente al operador > en SQL.

```
{"precio": {"$gt": 100}}
```

\$regex (Expresión Regular)

Permite búsquedas mediante patrones de texto. Útil para exfiltración carácter a carácter.

```
{"pass": {"$regex": "^\w{8}$"}}
```

Authentication Bypass en MongoDB

La Vulnerabilidad

Al explotar operadores como \$ne, podemos romper la lógica de autenticación y acceder como el primer usuario de la base de datos, típicamente el administrador.

Payload Malicioso

```
{  
  "username": {"$ne": null},  
  "password": {"$ne": null}  
}
```

Traducción del Ataque

"Dame cualquier usuario donde el nombre no sea nulo Y la contraseña no sea nula"

Resultado: El sistema retorna el **primer documento** que cumple la condición, usualmente la cuenta de administrador.



Inyección vía URL en PHP y Express

Los ataques NoSQL pueden ejecutarse directamente desde el navegador manipulando los parámetros de la URL. Los frameworks interpretan los corchetes como estructuras de datos.

Sintaxis del Ataque

```
login.php?user[$ne]=1&pass[$ne]=1
```

El backend interpreta automáticamente los `[$ne]` como un array u objeto JSON, pasándolo sin sanitizar a la consulta de MongoDB. Este comportamiento permite bypass completo de autenticación.

Exfiltración con Expresiones Regulares

1

Paso 1: Pregunta

¿La contraseña empieza con 'a'?

```
{"pass": {"$regex": "^\w{1}a\b"}}
```

2

Paso 2: Respuesta

El servidor responde "Login exitoso" o
error según coincida el patrón

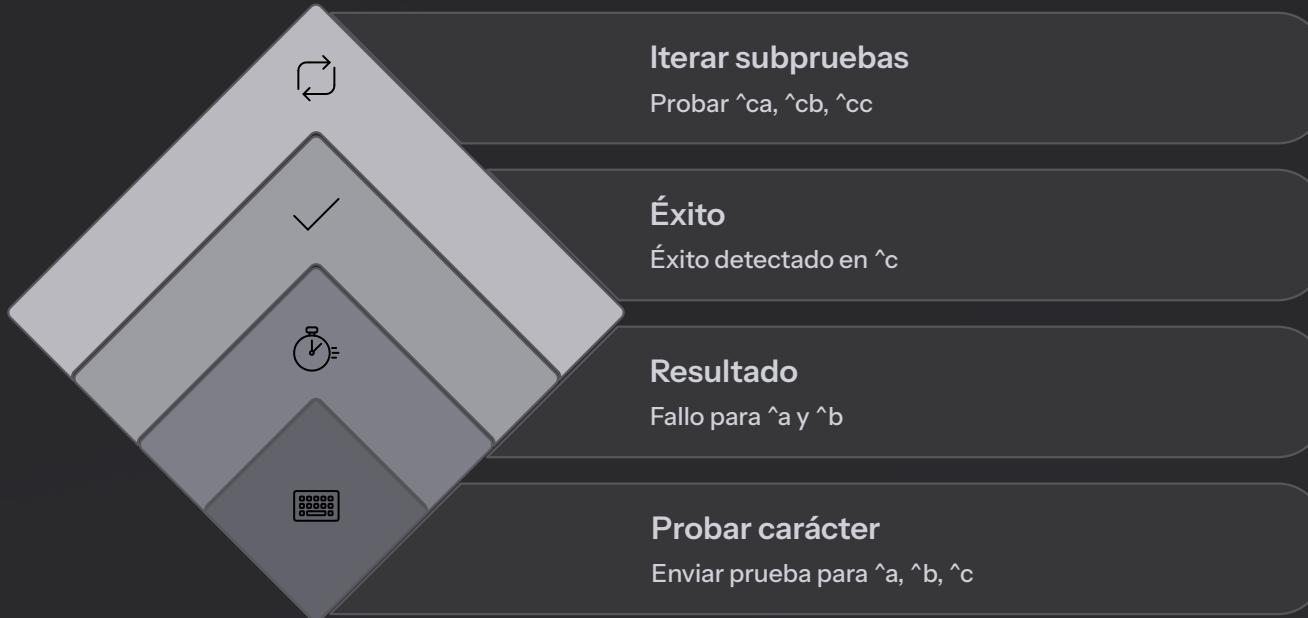
3

Paso 3: Confirmación

Primera letra confirmada. Continuar
con segundo carácter

El operador `$regex` permite adivinar datos mediante patrones de texto. Si el servidor confirma el patrón, hemos extraído información sensible carácter a carácter.

Blind NoSQL Injection: Enumeración Carácter a Carácter



Mecánica de Enumeración

Cuando no hay mensajes de error visibles, utilizamos técnicas *blind* para extraer información mediante respuestas binarias (verdadero/falso).

- Probar cada carácter: a , b , c ...
- Si ac tiene éxito, probar aca , acb ...
- Automatizar con scripts personalizados

Esta técnica requiere cientos de peticiones. Las herramientas de automatización son esenciales.

 ALTO RIESGO

JavaScript Injection: El Operador \$where

El operador `$where` es el más peligroso de MongoDB porque permite ejecutar código JavaScript puro directamente en el servidor de base de datos.

Ejemplo de Código Vulnerable

```
$where: "this.user == '' + input + """"
```

Denegación de Servicio

Inyectar bucles infinitos: '
`while(true){}`

Ejecución de Código

Ejecutar funciones del servidor si no
hay sandboxing adecuado

Exfiltración Avanzada

Acceder a otras colecciones
mediante lógica condicional
compleja

Defensa y Mitigación: Protección Efectiva



01

Casteo de Tipos

Forzar que todos los inputs sean cadenas de texto mediante `String(input)` antes de procesarlos

02

Sanitización

Implementar librerías especializadas como `mongo-sanitize` que eliminan operadores peligrosos

03

Validación con Esquemas

Usar Mongoose u ODMs similares con esquemas estrictos que rechacen objetos en campos de texto