**Exposición de Datos Sensibles**

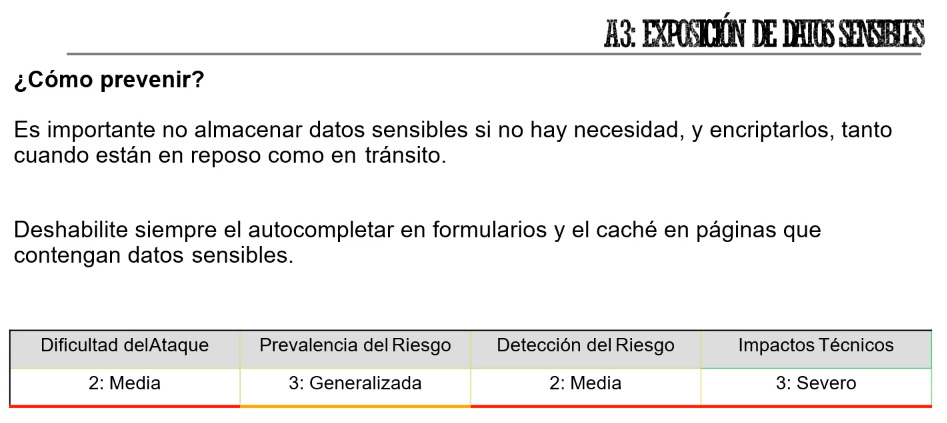


Este incidente se da en nuestra aplicación web cuando guardamos muchos datos que son propios del usuario.

Imaginemos un escenario en el que una aplicación web que dentro de la caché del navegador guardara el número de la tarjeta de crédito o incluso la contraseña. Eso sería evidentemente inaceptable. Luego, el hacker, por ejemplo, podría hacer una inyección por JavaScript y acceder a dichos datos guardado directamente en la caché de ese navegador y enviarlo a sí mismo o a una cuenta de correo.

Por lo tanto, tenemos que pensar que no podemos exponer dentro de nuestra aplicación web ningún dato sensible, ya que estos hackers podrían utilizarlo para robar o modificar datos en su propio beneficio o incluso robar a empresas o a otras personas.

Esto pasa mucho con fraudes de tarjetas de crédito, con suplantaciones de identidad, entre otros.



La solución a la exposición de datos sensibles es simple. Incluida la prevención sería no guardar los datos que no fueran necesarios. Además, es deseable que estos datos viajen de manera encriptado para que sea más complicado para el hacker descifrarlos.

También será necesario en los formularios de entrada de nuestras aplicaciones web deshabilitar la opción de autocompletado y no guardar nada en la caché de las páginas que sea realmente sensible.

**Deshabilitar la función de autocompletado en formularios**

Para deshabilitar la función de autocompletado en formularios HTML y evitar la exposición de datos sensibles, podemos usar el atributo autocomplete="off". Esto se puede aplicar tanto al elemento <form> como a campos individuales como <input>.

**Ejemplo:**

**1. Deshabilitar autocompletado en todo el formulario:**

<form autocomplete="off">

<input type="text" name="username" />

<input type="password" name="password" />

<button type="submit">Enviar</button>

</form>

**2. Deshabilitar autocompletado en un campo específico:**

<form>

<input type="text" name="username" autocomplete="off" />

<input type="password" name="password" autocomplete="off" />

<button type="submit">Enviar</button>

</form>

**Nota:**

* **Campos sensibles**: Es especialmente útil para campos como contraseñas, números de tarjeta de crédito o datos personales.
* **Compatibilidad**: La mayoría de los navegadores modernos respetan autocomplete="off", pero algunos (como Chrome) pueden ignorarlo en ciertos casos para mejorar la experiencia del usuario. Para mayor control, considera:
  + Usar type="password" para campos de contraseñas, ya que los navegadores suelen manejarlos de forma más segura.
  + Asignar nombres de campo dinámicos o aleatorios (por ejemplo, name="user\_123") para evitar que el navegador los reconozca como campos comunes.
  + Usar autocomplete="new-password" para campos de contraseñas nuevas, ya que algunos navegadores lo interpretan como un campo que no debe autocompletarse con datos almacenados.

**Ejemplo con new-password:**

<input type="password" name="newpassword" autocomplete="new-password" />

**Evitar almacenamiento en cache del navegador.**

<meta http-equiv="Cache-Control" content="no-store">

**Nota:**

* no-store prohíbe cualquier almacenamiento, incluso temporal, en caché del navegador, proxies o CDNs.
* Usar para recursos con datos sensibles, como formularios con información personal o contraseñas.

Imaginemos en un determinado ámbito que varios usuarios puedan tener acceso al mismo navegador web desde el que acceden a las aplicaciones y allí estén guardando, por ejemplo, dentro de la caché del navegador, la tarjeta de crédito. En ese caso, otro usuario que tenga acceso a ese equipo podría acceder a todos los datos de esa tarjeta de crédito.

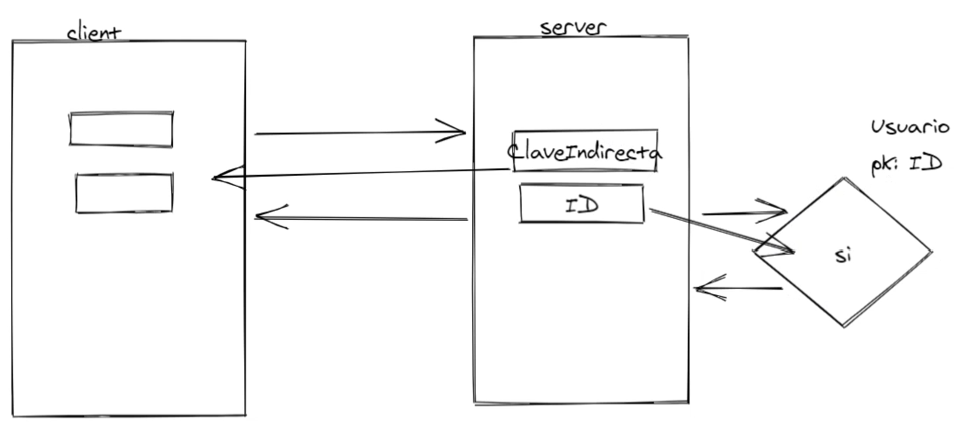
La dificultad de este ataque es media.

La prevalencia de riesgo es generalizada porque son muchos los que intentan acceder a los datos sensibles que están guardando nuestras aplicaciones web.

La detección de riesgo es media y de nuevo tenemos un impacto técnico severo, ya que una empresa o un usuario es muy perjudicado por este riesgo.

**Fundamento teórico de la Exposición de Datos**

En la figura podemos ver una arquitectura cliente servidor.



El servidor se conectará a un sistema de información, por ejemplo, una base de datos. Está bien que así sea ya que el cliente nunca debe acceder directo a la base.

Desde nuestro cliente tenemos un formulario con unos campos, para buscar un usuario en el servidor. El servidor buscara los datos en la base y devolverá el objeto solicitado por petición GET mediante un JSON, por ejemplo.

El problema reside en que si pasamos el ID desde el cliente al servidor, el hacker podría verlo y luego realizar una inyección SQL.

Podemos transformar y agregar capas de seguridad en el servidor utilizando el concepto de clave indirecta. La idea es que el cliente realice su búsqueda por dicha clave indirecta y no por el ID. Luego, en el servidor, se hace la conversión de la clave indirecta a él ID tradicional por el cual la aplicación web buscara los datos en la base relacionada. De esta manera, le ponemos una traba al hacker en su intento de acceder a los datos de la base. Lo normal es que este ID nunca salga de nuestro servidor, viajando en forma online y siempre estuviera protegido. Esto hace que la aplicación sea mucho más segura.

**Ejemplo: Búsqueda de un usuario con clave indirecta**

**Contexto**

Supongamos que tenemos una aplicación web donde los usuarios pueden buscar información de otros usuarios. En lugar de enviar el user\_id (por ejemplo, 123) directamente desde el cliente, generaremos una **clave indirecta** (un identificador opaco) que el servidor traducirá al user\_id internamente.

**1. Estructura de la base de datos**

Tenemos una tabla users y una tabla indirect\_keys para almacenar las claves indirectas asociadas a los IDs.

**Tabla users:**

| **user\_id** | **name** | **email** |
| --- | --- | --- |
| 123 | Juan Pérez | [juan@example.com](mailto:juan@example.com) |
| 124 | Ana Gómez | [ana@example.com](mailto:ana@example.com) |

**Tabla indirect\_keys:**

| **key\_id** | **user\_id** | **indirect\_key** | **created\_at** |
| --- | --- | --- | --- |
| 1 | 123 | abc123xyz | 2025-04-30 10:00:00 |
| 2 | 124 | def456uvw | 2025-04-30 10:01:00 |

* indirect\_key es un valor único generado (por ejemplo, un hash o UUID).
* created\_at puede usarse para establecer una expiración (opcional, para mayor seguridad).

**2. Lógica en el servidor**

**Tecnologías utilizadas (ejemplo):**

* Backend: Node.js con Express.
* Base de datos: PostgreSQL.
* Generación de claves indirectas: UUID o un hash seguro.

**Código del servidor (Node.js + Express)**

const express = require('express');

const { Pool } = require('pg');

const crypto = require('crypto');

const { v4: uuidv4 } = require('uuid');

const app = express();

app.use(express.json());

*// Conexión a PostgreSQL*

const pool = new Pool({

user: 'tu\_usuario',

host: 'localhost',

database: 'tu\_base\_de\_datos',

password: 'tu\_contraseña',

port: 5432,

});

*// Middleware para sanitizar entradas y prevenir inyección SQL*

const sanitizeInput = (input) => {

*// Aquí podrías usar una librería como 'sanitize-html' o regex para limpiar*

return input.replace(/[^a-zA-Z0-9-]/g, '');

};

*// Endpoint para obtener un usuario por clave indirecta*

app.get('/api/user/:indirectKey', async (req, res) => {

const indirectKey = sanitizeInput(req.params.indirectKey);

try {

*// Buscar el user\_id asociado a la clave indirecta*

const keyResult = await pool.query(

'SELECT user\_id FROM indirect\_keys WHERE indirect\_key = $1',

[indirectKey]

);

if (keyResult.rows.length === 0) {

return res.status(404).json({ error: 'Clave indirecta no válida' });

}

const userId = keyResult.rows[0].user\_id;

*// Buscar los datos del usuario en la tabla users*

const userResult = await pool.query(

'SELECT name, email FROM users WHERE user\_id = $1',

[userId]

);

if (userResult.rows.length === 0) {

return res.status(404).json({ error: 'Usuario no encontrado' });

}

*// Devolver los datos del usuario (sin exponer el user\_id)*

res.json(userResult.rows[0]);

} catch (err) {

console.error(err);

res.status(500).json({ error: 'Error en el servidor' });

}

});

*// Endpoint para generar una clave indirecta para un usuario (simulado)*

app.post('/api/generate-key', async (req, res) => {

const { user\_id } = req.body;

*// Validar que user\_id existe*

const userResult = await pool.query(

'SELECT user\_id FROM users WHERE user\_id = $1',

[user\_id]

);

if (userResult.rows.length === 0) {

return res.status(404).json({ error: 'Usuario no encontrado' });

}

*// Generar clave indirecta (por ejemplo, UUID)*

const indirectKey = uuidv4();

*// Guardar la clave indirecta en la base de datos*

await pool.query(

'INSERT INTO indirect\_keys (user\_id, indirect\_key, created\_at) VALUES ($1, $2, NOW())',

[user\_id, indirectKey]

);

*// Devolver la clave indirecta al cliente*

res.json({ indirect\_key: indirectKey });

});

app.listen(3000, () => {

console.log('Servidor corriendo en puerto 3000');

});

**3. Flujo en el cliente**

**Paso 1: Generar una clave indirecta**

El cliente (por ejemplo, una aplicación frontend) solicita una clave indirecta para un user\_id específico. Esto podría hacerse en un contexto autenticado, donde el servidor valida que el usuario tiene permiso para acceder a ese user\_id.

**Solicitud:**

POST /api/generate-key

Content-Type: application/json

{

"user\_id": 123

}

**Respuesta:**

{

"indirect\_key": "abc123xyz"

}

**Paso 2: Buscar un usuario con la clave indirecta**

El cliente usa la clave indirecta para buscar los datos del usuario, sin exponer el user\_id.

**Solicitud:**

GET /api/user/abc123xyz

**Respuesta:**

{

"name": "Juan Pérez",

"email": "juan@example.com"

}

**4. Seguridad adicional**

1. **Sanitización de entradas**: El servidor valida y limpia la indirect\_key para prevenir inyección SQL u otros ataques.
2. **Expiración de claves indirectas**: Puedes agregar una columna expires\_at en la tabla indirect\_keys y verificar que la clave no haya expirado.
3. **Autenticación y autorización**: Asegúrate de que solo usuarios autenticados puedan generar o usar claves indirectas.
4. **Claves únicas y seguras**: Usa UUID o un hash (por ejemplo, SHA-256) para generar claves indirectas impredecibles.
5. **HTTPS**: Todas las comunicaciones entre cliente y servidor deben usar HTTPS para evitar que las claves indirectas sean interceptadas.
6. **Limitar intentos**: Implementa un sistema de rate-limiting para prevenir ataques de fuerza bruta contra las claves indirectas.

**5. Ventajas de este enfoque**

* **Oculta el ID**: El user\_id nunca se expone al cliente, reduciendo el riesgo de inyección SQL o enumeración de IDs.
* **Capas de abstracción**: La clave indirecta actúa como un proxy, dificultando el acceso directo a la base de datos.
* **Flexibilidad**: Puedes invalidar o regenerar claves indirectas sin afectar los IDs reales.
* **Auditoría**: Puedes rastrear el uso de claves indirectas para detectar comportamientos sospechosos.

**6. Limitaciones**

* **Gestión de claves**: Necesitas mantener la tabla indirect\_keys actualizada y limpiar claves expiradas.
* **Rendimiento**: La consulta adicional para traducir la clave indirecta al user\_id puede añadir una pequeña sobrecarga.
* **Complejidad**: Agrega una capa adicional de lógica al sistema.

**UUID (Universally Unique Identifier)**

Un **UUID** (Universally Unique Identifier, Identificador Único Universal) es un estándar para generar identificadores únicos de 128 bits, representados generalmente como una cadena de 36 caracteres en formato hexadecimal, dividida en cinco grupos separados por guiones. Por ejemplo:

550e8400-e29b-41d4-a716-446655440000

**Características principales del UUID**

1. **Unicidad**: Está diseñado para ser único en todo el mundo, incluso sin coordinación centralizada, gracias a su gran espacio de posibles valores (2^128, aproximadamente 3.4x10^38 combinaciones).
2. **Estandarizado**: Definido por la norma RFC 4122, lo que garantiza compatibilidad entre sistemas.
3. **Formato**: Se representa como una cadena de 8-4-4-4-12 caracteres hexadecimales (por ejemplo, 123e4567-e89b-12d3-a456-426614174000).
4. **Generación descentralizada**: Puede generarse en cualquier sistema sin necesidad de una autoridad central, usando algoritmos que combinan factores como el tiempo, dirección MAC, números aleatorios, etc.

**Versiones de UUID**

Hay varias versiones de UUID, cada una con un método de generación diferente:

* **Versión 1**: Basada en la marca de tiempo (timestamp) y la dirección MAC del dispositivo. Ejemplo: 550e8400-e29b-11d4-a716-446655440000.
* **Versión 2**: Similar a la versión 1, pero incluye información de dominio (menos común).
* **Versión 3**: Generada a partir de un namespace y un nombre, usando un hash MD5. Ejemplo: determinística para los mismos datos de entrada.
* **Versión 4**: Generada de forma aleatoria o pseudoaleatoria, la más común para aplicaciones modernas. Ejemplo: 123e4567-e89b-4d3-a456-426614174000.
* **Versión 5**: Similar a la versión 3, pero usa un hash SHA-1, más seguro.

**Usos comunes**

* **Claves indirectas**: Como en el ejemplo que mencionaste, para ocultar IDs sensibles en aplicaciones web.
* **Identificadores en bases de datos**: Para garantizar unicidad sin exponer IDs secuenciales.
* **Sistemas distribuidos**: Para identificar recursos en entornos donde múltiples nodos generan datos simultáneamente.
* **APIs y microservicios**: Como identificadores de sesiones, transacciones o recursos.

**Ventajas**

* **Unicidad global**: Muy baja probabilidad de colisiones, incluso en sistemas distribuidos.
* **Descentralizado**: No requiere un servidor central para generarlos.
* **Seguridad relativa**: En la versión 4, la aleatoriedad dificulta la predicción de valores.

**Desventajas**

* **Tamaño**: Más grande que un ID numérico (36 caracteres vs. un entero).
* **Legibilidad**: No es amigable para humanos.
* **Rendimiento**: En bases de datos, puede ser menos eficiente que IDs secuenciales para índices.

**Ejemplo**

En el ejemplo del servidor Node.js que te di, usé la librería uuid para generar una clave indirecta con v4 (aleatoria):

const { v4: uuidv4 } = require('uuid');

const indirectKey = uuidv4(); *// Genera algo como "550e8400-e29b-41d4-a716-446655440000"*

Esta clave (indirectKey) se almacena en la tabla indirect\_keys y se usa para buscar un user\_id sin exponerlo al cliente, añadiendo una capa de seguridad.

**Implementación en otros lenguajes**

* **Python**: import uuid; print(uuid.uuid4())
* **Java**: UUID.randomUUID().toString()
* **PHP**: Ramsey\Uuid\Uuid::uuid4()->toString()
* **Ruby**: SecureRandom.uuid