UNIVERSIDAD DEL ISTMO
FACULTAD DE INGENIERÍA

PROYECTO 1
SQLi

CARLOS SOLARES

Guatemala, 20 de Agosto del 2024

1. Introducción

1.1. Objetivo del Proyecto

Descripción:

El objetivo principal de este proyecto es demostrar y entender la vulnerabilidad de SQL Injection (SQLi) en aplicaciones web. SQL Injection es una técnica de ataque que explota vulnerabilidades en las aplicaciones web para ejecutar consultas SQL maliciosas. Este proyecto se centra en dos aspectos clave:

1. **Demostración de Vulnerabilidad:** Crear un entorno controlado donde se pueda observar cómo un atacante puede explotar una vulnerabilidad de SQL Injection para extraer datos, modificar estructuras de bases de datos, y ejecutar otros comandos SQL no autorizados.
2. **Implementación de Medidas de Protección:** Una vez que se han demostrado las vulnerabilidades, se implementarán técnicas de protección para mitigar los riesgos asociados con SQL Injection. Esto incluye el uso de consultas parametrizadas, validación de entradas y otros métodos para asegurar la integridad y seguridad de la aplicación.

2. Arquitectura del Sistema

2.1. Diagrama del Sistema

Descripción del Diagrama:

Un diagrama de arquitectura del sistema proporciona una vista general de cómo los componentes del sistema interactúan entre sí. En este proyecto, el diagrama debe incluir los siguientes componentes:

1. **Frontend:**
    - **Formulario Web:** Representa la interfaz de usuario que permite a los usuarios ingresar sus credenciales. Este formulario envía solicitudes al backend.
2. **Backend:**
    - **Servidor Node.js:** El servidor que recibe y procesa las solicitudes del formulario web. Implementa la lógica de negocio y las consultas a la base de datos.
    - **Gestión de SQL Injection:** Parte del código del backend que es vulnerable y la que ha sido modificada para proteger contra SQL Injection.
3. **Base de Datos:**

- **SQLite:** La base de datos utilizada para almacenar usuarios y contraseñas. La base de datos está en memoria durante la ejecución del proyecto.

2.2. Descripción del Sistema

Frontend:

1. **Formulario Web:**
   - **Funcionalidad:** El frontend consiste en un formulario HTML que permite a los usuarios ingresar su nombre de usuario y contraseña. Estos datos se envían al servidor mediante una solicitud POST.
   - **Componentes:**
     - **Campo de Usuario:** Un campo de texto donde los usuarios ingresan su nombre de usuario.
     - **Campo de Contraseña:** Un campo de entrada para la contraseña.
     - **Botón de Envío:** Envía los datos del formulario al backend para su procesamiento.

html
Copy code
```
<form method="POST" action="/login">
    <label>Usuario:</label>
    <input type="text" name="username" />
    <label>Contraseña:</label>
    <input type="password" name="password" />
    <button type="submit">Iniciar Sesión</button>
</form>
```

2.

Backend:

1. **Servidor Node.js:**
   - **Funcionalidad:** El servidor Node.js recibe las solicitudes del formulario, ejecuta consultas SQL en la base de datos SQLite, y devuelve una respuesta al usuario.
   - **Componentes:**
     - **Rutas de HTTP:** Define las rutas para manejar las solicitudes, en este caso, la ruta POST `/login`.
     - **Lógica de Consultas SQL:** Contiene las consultas SQL vulnerables y protegidas para demostrar SQL Injection y prevención.
     - **Manejo de Errores:** Captura y maneja errores en la base de datos, incluyendo los errores provocados por SQL Injection.

Código Ejemplo (Vulnerable):
javascript

Copy code

```
app.post('/login', (req, res) => {
    const { username, password } = req.body;
    const query = `SELECT username FROM users WHERE username =
'${username}' AND password = '${password}' UNION SELECT 'Bienvenido,
' || username || ' - ' || password, '' FROM users`;

    db.all(query, [], (err, rows) => {
        if (err) {
            return res.send(`Error en la base de datos:
${err.message}`);
        }
        if (rows.length > 0) {
            res.send(`Bienvenido, ${rows[0].username}`);
        } else {
            res.send('Usuario o contraseña incorrectos');
        }
    });
});
```

2.

Base de Datos:

1. **SQLite:**
    - **Funcionalidad:** SQLite es una base de datos ligera y en memoria utilizada para almacenar la información de usuarios y contraseñas en este proyecto. Es ideal para demostraciones y pruebas debido a su simplicidad y facilidad de configuración.
    - **Estructura:**
        - **Tabla de Usuarios:** Una tabla que contiene columnas para `id`, `username`, y `password`.

Código de Creación de la Tabla:
javascript
Copy code

```
db.serialize(() => {
    db.run(`CREATE TABLE users (id INTEGER PRIMARY KEY, username
TEXT, password TEXT)`);
    db.run(`INSERT INTO users (username, password) VALUES ('admin',
'admin123')`);
});
```

2. **Descripción de las Columnas:**
    - **id:** Identificador único para cada usuario.

- **username:** Nombre de usuario.
- **password:** Contraseña del usuario.

2.3. Flujo de Datos

1. **Ingreso de Datos:**
   - El usuario ingresa sus credenciales en el formulario web y hace clic en "Iniciar Sesión".
2. **Procesamiento en el Backend:**
   - El servidor Node.js recibe los datos del formulario.
   - Ejecuta una consulta SQL utilizando los datos proporcionados.
3. **Interacción con la Base de Datos:**
   - La consulta SQL se envía a SQLite, que la procesa y devuelve los resultados al servidor.
4. **Respuesta al Usuario:**
   - El servidor Node.js recibe los resultados de la base de datos y envía una respuesta al usuario, que puede ser un mensaje de bienvenida o un error.

3. Código y Configuración

3.1. Backend

Descripción del Código:

El backend del proyecto está desarrollado en Node.js y utiliza el framework Express para manejar solicitudes HTTP. La base de datos utilizada es SQLite, que se configura en memoria para fines de prueba. A continuación, se detalla el código relevante y su configuración.

Código del Servidor Node.js:

1. **Configuración Inicial:**
   - **Instalación de Dependencias:** Asegúrate de tener las dependencias necesarias instaladas mediante npm (`express` y `sqlite3`).

```
npm install express cross-env sqlite3
```

2. **Estructura del Proyecto:**

```
/project-directory

├── index.js
├── protected.js
├── vulnerable.js
├── package.json
```

└── README.md

-

Código del Servidor Node.js (index.js):
Código Vulnerable: Este código demuestra la vulnerabilidad de SQL Injection mediante una consulta SQL construida con concatenación de cadenas.

```
const http = require('http');
const mode = process.env.MODE || 'vulnerable';
let app;

if (mode === 'protected') {
    app = require('./protected');
} else {
    app = require('./vulnerable');
}

const server = http.createServer(app);

server.listen(3000, () => {
            console.log(`Servidor iniciado en modo ${mode} en
http://localhost:3000`);
});
```

3. **Descripción del Código Vulnerable:**

```
const express = require('express');

const sqlite3 = require('sqlite3').verbose();

const app = express();



const db = new sqlite3.Database(':memory:');



app.use(express.urlencoded({ extended: true }));



app.get('/', (req, res) => {
```

```
    res.send(`
        <form method="POST" action="/login">
            <label>Usuario:</label>
            <input type="text" name="username" />
            <label>Contraseña:</label>
            <input type="password" name="password" />
            <button type="submit">Iniciar Sesión</button>
        </form>
    `);
});


app.post('/login', (req, res) => {
    const { username, password } = req.body;

    // Consulta SQL vulnerable
    const query = `SELECT username FROM users WHERE username =
'${username}' AND password = '${password}'`;

    db.all(query, [], (err, rows) => {
        if (err) {
            return res.send(`Error en la base de datos:
${err.message}`);
        }
        if (rows.length > 0) {
            const result = rows.map(row => `Bienvenido,
${row.username}`).join('<br/>');
```

```
            res.send(result);
        } else {
            res.send('Usuario o contraseña incorrectos');
        }
    });
});

db.serialize(() => {
    db.run(`CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT, password TEXT)`);
    db.run(`INSERT INTO users (username, password) VALUES ('admin', 'admin123')`);
    db.run(`INSERT INTO users (username, password) VALUES ('user1', 'password1')`);
    db.run(`INSERT INTO users (username, password) VALUES ('user2', 'password2')`);
    db.run(`INSERT INTO users (username, password) VALUES ('user3', 'password3')`);
});

module.exports = app;
```

- **Consulta SQL Vulnerable:** La consulta SQL utilizada en la ruta `/login` es vulnerable a inyecciones SQL. Permite a un atacante inyectar comandos SQL maliciosos debido a la concatenación directa de las variables `username` y `password`.
- **Union SQL Injection:** La parte de la consulta con `UNION SELECT` es utilizada para inyectar comandos SQL adicionales que permiten extraer información no autorizada de la base de datos.

Código Protegido: En el modo protegido, se utiliza una consulta parametrizada para evitar la vulnerabilidad.

Código del Servidor Node.js (protected.js):

```
const http = require('http');
const mode = process.env.MODE || 'vulnerable';
let app;

if (mode === 'protected') {
    app = require('./protected');
} else {
    app = require('./vulnerable');
}

const server = http.createServer(app);

server.listen(3000, () => {
            console.log(`Servidor iniciado en modo ${mode} en
http://localhost:3000`);
});
```

4. **Descripción del Código Protegido:**
    - **Consulta SQL Protegida:** Utiliza una consulta parametrizada (?) en lugar de concatenar las variables directamente. Esto previene la inyección de código SQL malicioso al tratar las variables como datos y no como parte de la consulta SQL.

3.2. Base de Datos

Descripción de la Base de Datos SQLite:

SQLite es una base de datos ligera que se almacena en memoria para fines de prueba. Su configuración es sencilla y se utiliza para almacenar datos de usuarios en este proyecto.

1. **Configuración Inicial:**
    - **Base de Datos en Memoria:** Se utiliza `':memory:'` para crear una base de datos temporal que reside en la memoria durante la ejecución del servidor.
2. **Creación de la Tabla de Usuarios:**

Código para Crear la Tabla:

```
db.serialize(() => {
    db.run(`CREATE TABLE users (id INTEGER PRIMARY KEY, username TEXT,
password TEXT)`);
```

```
      db.run(`INSERT INTO users (username, password) VALUES ('admin',
'admin123')`);
      db.run(`INSERT INTO users (username, password) VALUES ('user1',
'password1')`);
      db.run(`INSERT INTO users (username, password) VALUES ('user2',
'password2')`);
      db.run(`INSERT INTO users (username, password) VALUES ('user3',
'password3')`);
});
```

- **Descripción de la Tabla:**
  - **id:** Campo entero que actúa como clave primaria, identificador único para cada usuario.
  - **username:** Campo de texto que almacena el nombre de usuario.
  - **password:** Campo de texto que almacena la contraseña del usuario.

3.3. Cambios y Configuración

Configuración del Entorno:

- **Archivos de Configuración:** Asegúrate de tener un archivo `package.json` que liste todas las dependencias necesarias y los scripts para ejecutar el servidor.

Pruebas Iniciales:

- **Verificación de Funcionamiento Básico:** Asegúrate de que el servidor se inicia correctamente y que la base de datos se crea y se llena con datos de prueba. Realiza pruebas básicas para verificar que el formulario web funciona y que las solicitudes POST se manejan correctamente.

4. SQL Injection y Prevención

4.1. Explicación de SQL Injection

Qué es SQL Injection:

SQL Injection (SQLi) es una técnica de ataque en la que un atacante inserta o "inyecta" código SQL malicioso en una consulta SQL legítima que es ejecutada por una aplicación web. Esta vulnerabilidad ocurre cuando una aplicación permite a los

usuarios enviar datos que se incluyen directamente en una consulta SQL sin una validación adecuada.

Cómo Funciona:

1. **Entrada del Usuario:** El atacante envía datos a través de un formulario web, una URL o una API que son utilizados en una consulta SQL.
2. **Inyección de Código Malicioso:** El atacante incluye código SQL en los datos de entrada que se incorporan directamente en la consulta SQL ejecutada por la base de datos.
3. **Ejecución en la Base de Datos:** La base de datos ejecuta el código malicioso junto con la consulta legítima, permitiendo al atacante realizar acciones no autorizadas como leer, modificar o eliminar datos.

Ejemplo de SQL Injection:

Supongamos que una aplicación web tiene un formulario de inicio de sesión vulnerable a SQL Injection:

Consulta SQL Vulnerable:

sql

Copy code

```
SELECT * FROM users WHERE username = 'input_username' AND
password = 'input_password';
```

Si el atacante ingresa `' OR '1'='1` como nombre de usuario y contraseña, la consulta resultante será:

sql

Copy code

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND
password = '';
```

La condición `'1'='1'` siempre es verdadera, por lo que el atacante puede acceder al sistema sin credenciales válidas.

4.2. Técnicas de Prevención Contra SQL Injection

1. Uso de Consultas Parametrizadas

Descripción: Las consultas parametrizadas (o preparadas) son una técnica para evitar SQL Injection al separar el código SQL de los datos proporcionados por el usuario. Las consultas parametrizadas tratan los datos de entrada como parámetros en lugar de parte de la consulta SQL.

Ejemplo de Consulta Parametrizada:

Código Protegido:

javascript

Copy code

```
const query = `SELECT username FROM users WHERE username = ?
AND password = ?`;

db.all(query, [username, password], (err, rows) => {

    if (err) {

        return res.send(`Error en la base de datos:
${err.message}`);

    }

    if (rows.length > 0) {

        res.send(`Bienvenido, ${rows[0].username}`);

    } else {

        res.send('Usuario o contraseña incorrectos');

    }

});
```

Beneficios:

- **Separación de Datos y Código:** Los parámetros se envían por separado y se insertan en la consulta en el momento de la ejecución.

- **Prevención Automática:** El motor de base de datos maneja el escape de caracteres especiales en los parámetros.

2. Validación y Saneamiento de Entradas

Descripción: Validar y sanear los datos de entrada asegura que sólo se acepten valores válidos y seguros. Esto incluye verificar el tipo, formato y longitud de los datos ingresados por el usuario.

Ejemplo de Validación:

javascript

Copy code

```
const username = req.body.username;

const password = req.body.password;


// Validar longitud y formato

if (username.length > 50 || password.length > 50) {

    return res.send('Datos de entrada demasiado largos');

}


// Más validaciones según los requisitos de la aplicación
```

Beneficios:

- **Reducción de Superficie de Ataque:** Limita los tipos de datos que se pueden enviar, reduciendo la posibilidad de inyecciones maliciosas.
- **Seguridad Adicional:** Previene otros tipos de ataques relacionados con la entrada de datos.

3. Uso de ORM (Object-Relational Mapping)

Descripción: Los ORMs como Sequelize (para Node.js) abstraen las consultas SQL en métodos de alto nivel, reduciendo el riesgo de SQL Injection al no permitir la construcción manual de consultas SQL.

Ejemplo de ORM:

Código con Sequelize:

javascript

Copy code

```
const User = require('./models/user'); // Modelo de usuario
con Sequelize


User.findOne({
    where: {
        username: username,
        password: password
    }
}).then(user => {
    if (user) {
        res.send(`Bienvenido, ${user.username}`);
    } else {
        res.send('Usuario o contraseña incorrectos');
    }
});
```

Beneficios:

- **Abstracción de SQL:** El ORM maneja las consultas SQL internamente, evitando la exposición a inyecciones.
- **Facilidad de Uso:** Ofrece una interfaz de alto nivel para interactuar con la base de datos.

4. Configuración Segura del Sistema de Base de Datos

Descripción: Configurar la base de datos de manera segura incluye limitar los permisos de usuario, auditar consultas y utilizar roles específicos para diferentes tipos de operaciones.

Ejemplo de Configuración:

- **Uso de Cuentas con Privilegios Mínimos:** Crear cuentas de base de datos con los permisos mínimos necesarios para la aplicación.
- **Auditoría de Consultas:** Habilitar la auditoría de consultas para detectar comportamientos sospechosos.

Beneficios:

- **Mitigación de Daños:** Reduce el impacto de un ataque SQL Injection al limitar las acciones que un atacante puede realizar.
- **Monitoreo:** Permite identificar y responder rápidamente a ataques potenciales.

4.3. Repercusiones de No Implementar Medidas de Prevención

Impacto en la Seguridad:

- **Exposición de Datos Sensibles:** Los atacantes pueden acceder a información confidencial como credenciales de usuario y datos personales.
- **Modificación o Eliminación de Datos:** Permite la alteración o eliminación de datos críticos en la base de datos.
- **Compromiso Total del Sistema:** En casos graves, un atacante podría obtener el control total del servidor de base de datos.

Ejemplos de Incidentes Reales:

- **Ataques a Empresas:** Casos documentados de empresas que sufrieron pérdidas financieras y de reputación debido a vulnerabilidades de SQL Injection.
- **Exposición de Datos:** Ejemplos de filtraciones masivas de datos personales debido a fallos en la seguridad de aplicaciones web.

5. Conclusión

5.1. Resumen del Proyecto

En este proyecto, hemos desarrollado un sitio web simple con un formulario de inicio de sesión para demostrar la vulnerabilidad de SQL Injection y cómo protegerse contra ella. Utilizando un backend en Node.js con una base de datos SQLite en memoria, hemos mostrado tanto un modo vulnerable como uno protegido, enfatizando la importancia de la seguridad en el desarrollo de aplicaciones web.

5.2. Principales Hallazgos

1. **Vulnerabilidad de SQL Injection:**
    - La consulta SQL vulnerable permitió la inyección de código malicioso a través de la entrada del usuario. La técnica de SQL Injection demostró cómo un atacante puede manipular consultas SQL para obtener información no autorizada, modificar datos o incluso comprometer el sistema.
2. **Medidas de Prevención:**
    - **Consultas Parametrizadas:** La implementación de consultas parametrizadas demostró ser una solución efectiva para prevenir SQL Injection, al tratar los datos de entrada como parámetros en lugar de parte del código SQL.
    - **Validación y Saneamiento de Entradas:** Validar y sanear las entradas del usuario ayuda a asegurar que los datos sean de un formato esperado y seguro.
    - **Uso de ORMs:** El uso de Object-Relational Mappers (ORMs) proporciona una capa adicional de seguridad al abstraer la construcción de consultas SQL.
    - **Configuración Segura de la Base de Datos:** Configurar adecuadamente el sistema de base de datos y limitar los privilegios de usuario puede reducir el impacto de posibles ataques.

5.3. Impacto y Relevancia

La demostración de SQL Injection y la implementación de técnicas de prevención subrayan la importancia de asegurar las aplicaciones web contra vulnerabilidades comunes. SQL Injection sigue siendo una de las amenazas más prevalentes y peligrosas en el desarrollo de software, y su prevención es crucial para proteger la integridad y confidencialidad de los datos.

5.4. Recomendaciones

1. **Implementar Seguridad desde el Inicio:**
    - Es fundamental integrar medidas de seguridad en las primeras etapas del desarrollo de software, en lugar de tratar de remediar vulnerabilidades en fases posteriores.
2. **Educación y Capacitación:**
    - Los desarrolladores deben estar capacitados en las mejores prácticas de seguridad y en el manejo adecuado de datos de entrada para prevenir vulnerabilidades.
3. **Monitoreo y Auditoría:**
    - Realizar auditorías periódicas y monitorear el sistema para detectar y responder a posibles ataques puede ayudar a mantener la seguridad a largo plazo.
4. **Actualización Continua:**
    - Mantenerse al tanto de las últimas amenazas y vulnerabilidades, y actualizar las prácticas de seguridad y las herramientas en consecuencia, es esencial para proteger las aplicaciones web.

5.5. Reflexión Final

La seguridad informática es una responsabilidad crítica para todos los desarrolladores y administradores de sistemas. La capacidad de identificar y mitigar vulnerabilidades como SQL Injection no solo protege los datos y la infraestructura, sino que también contribuye a la confianza del usuario y la integridad del sistema. Este proyecto ha demostrado cómo las técnicas de prevención adecuadas pueden hacer una diferencia significativa en la seguridad de las aplicaciones web.