Sesión 6: Pruebas de seguridad y protección contra ataques

Duración: 45 minutos

🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

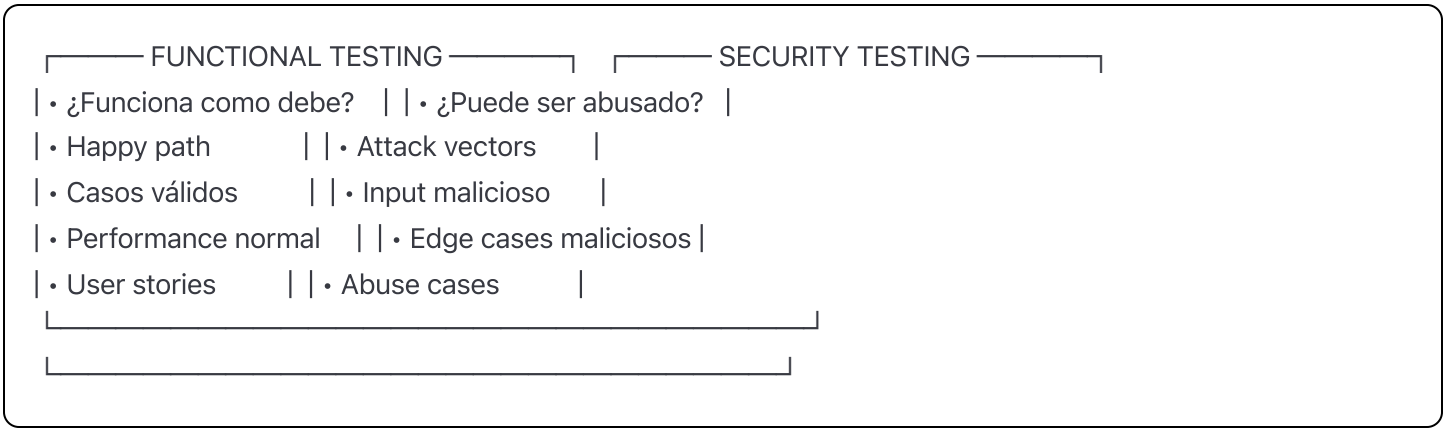
- Implementar pruebas automatizadas de seguridad en aplicaciones Python
- Identificar y prevenir las vulnerabilidades del OWASP Top 10
- Crear test suites para autenticación y autorización
- Usar herramientas especializadas para security testing
- Configurar pipelines de seguridad automatizada
- Simular ataques comunes para validar defensas

📋 Contenido

1. Security Testing vs Functional Testing (8 min)

Analogía: Si el testing funcional verifica que las cerraduras funcionen correctamente, el security testing verifica que no haya puertas traseras, ventanas abiertas, o formas de hacer ganzúa a las cerraduras.

Diferencias fundamentales:



Tipos de Security Testing:

```
🔒 AUTHENTICATION TESTING
├── Credential validation
├── Session management
├── Multi-factor authentication
└── Password policies

🛡️ AUTHORIZATION TESTING
├── Role-based access control
├── Permission boundaries
├── Privilege escalation
└── Resource access limits

⚔️ VULNERABILITY TESTING
├── Input validation attacks
├── Injection vulnerabilities
├── Business logic flaws
└── Configuration issues

🔍 PENETRATION TESTING
├── Automated vulnerability scans
├── Manual security testing
├── Social engineering simulation
└── Infrastructure testing
```

Shift-Left Security:

- **Developer testing** → Security unit tests
- **CI/CD integration** → Automated scans
- **Code review** → Security-focused reviews
- **Threat modeling** → Design-time security

2. OWASP Top 10 en Contexto de Testing (10 min)

Los riesgos más críticos y cómo testearlos:

1. Broken Access Control

```python
```

```python
# Test case ejemplo
def test_unauthorized_access_prevention():
    # Usuario sin permisos intenta acceder a recurso admin
    response = client.get('/admin/users', headers={'Authorization': 'Bearer user_token'})
    assert response.status_code == 403
    assert 'insufficient privileges' in response.json()['error']
```

2. Cryptographic Failures

```python
def test_password_hashing_security():
    # Verificar que passwords no se almacenen en texto plano
    user = create_user('test@example.com', 'plaintext_password')
    assert user.password_hash != 'plaintext_password'
    assert len(user.password_hash) > 50  # Hash debe ser largo
    assert '$' in user.password_hash  # Formato de hash seguro
```

3. Injection Attacks

```python
def test_sql_injection_prevention():
    malicious_input = "'; DROP TABLE users; --"
    response = client.get(f'/search?q={malicious_input}')
    # No debe crashear y no debe ejecutar SQL malicioso
    assert response.status_code in [200, 400]
    # Verificar que tabla users sigue existiendo
    assert User.query.count() > 0
```

4. Insecure Design

```python
```

```python
def test_business_logic_abuse():
    # Intentar transferir más dinero del disponible
    response = client.post('/transfer', json={
        'from_account': 'user1',
        'to_account': 'user2',
        'amount': 99999999  # Cantidad excesiva
    })
    assert response.status_code == 400
    assert 'insufficient funds' in response.json()['error']
```

5. Security Misconfiguration

```python
def test_debug_mode_disabled_in_production():
    # Verificar que debug está deshabilitado
    response = client.get('/non-existent-endpoint')
    assert response.status_code == 404
    # No debe revelar stack traces o info del sistema
    assert 'Traceback' not in response.text
    assert 'DEBUG' not in response.text
```

3. Test Cases de Autenticación y Autorización (8 min)

Authentication Test Patterns:

```python
```

```python
class TestAuthentication:
    def test_login_with_valid_credentials(self):
        """Happy path - credenciales válidas"""

    def test_login_with_invalid_password(self):
        """Password incorrecto debe fallar"""

    def test_login_with_nonexistent_user(self):
        """Usuario que no existe debe fallar"""

    def test_brute_force_protection(self):
        """Múltiples intentos fallidos deben bloquear"""

    def test_session_expiration(self):
        """Sesiones deben expirar después de tiempo límite"""

    def test_concurrent_session_limits(self):
        """Límite de sesiones simultáneas"""

    def test_logout_invalidates_session(self):
        """Logout debe invalidar tokens/sesiones"""
```

Authorization Test Patterns:

```python
class TestAuthorization:
    def test_admin_only_endpoints(self):
        """Solo admins pueden acceder a endpoints admin"""

    def test_user_can_only_access_own_data(self):
        """Usuarios solo ven sus propios datos"""

    def test_role_inheritance(self):
        """Jerarquía de roles funciona correctamente"""

    def test_permission_boundaries(self):
        """Permisos específicos se respetan"""

    def test_privilege_escalation_prevention(self):
        """Usuarios no pueden elevarse privilegios"""
```

Session Management Testing:

```python
def test_session_security():
    # Login y obtener session token
    login_response = client.post('/login', json=credentials)
    token = login_response.json()['token']

    # Token debe ser suficientemente largo y random
    assert len(token) >= 32
    assert token.isalnum() == False  # Debe tener caracteres especiales

    # Session debe expirar
    time.sleep(SESSION_TIMEOUT + 1)
    response = client.get('/profile', headers={'Authorization': f'Bearer {token}'})
    assert response.status_code == 401
```

4. Input Validation y Sanitization Testing (7 min)

Estrategias de input malicioso:

```
🚫 INJECTION PAYLOADS:
├── SQL: ' OR '1'='1' --
├── NoSQL: {"$ne": null}
├── LDAP: *)(uid=*))(|(uid=*
├── Command: ; cat /etc/passwd
└── Python: __import__('os').system('rm -rf /')

🔍 XSS PAYLOADS:
├── Stored: <script>alert('XSS')</script>
├── Reflected: <img src=x onerror=alert(1)>
├── DOM: javascript:alert(document.cookie)
└── Polyglot: jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert()
)//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert()//>

⚡ SPECIAL CHARACTERS:
├── Null bytes: \0, %00
├── Unicode:  script , \u003cscript\u003e
├── Encoding: %3Cscript%3E, &lt;script&gt;
└── Path traversal: ../../../etc/passwd
```

Test cases para input validation:

```python
class TestInputValidation:
    @pytest.mark.parametrize("malicious_input", [
        "'; DROP TABLE users; --",          # SQL injection
        "<script>alert('xss')</script>",    # XSS
        "../../../../etc/passwd",           # Path traversal
        "\x00\x01\x02",                     # Null bytes
        "A" * 10000,                        # Buffer overflow attempt
        {"$ne": None},                      # NoSQL injection
    ])
    def test_malicious_input_rejection(self, malicious_input):
        """Todos los inputs maliciosos deben ser rechazados"""
        response = client.post('/api/search', json={'query': malicious_input})
        assert response.status_code in [400, 422]  # Bad request o validation error

    def test_html_sanitization(self):
        """HTML malicioso debe ser sanitizado"""
        malicious_html = "<script>alert('xss')</script><b>Bold text</b>"
        response = client.post('/api/comment', json={'content': malicious_html})

        # Debe aceptar el request
        assert response.status_code == 201

        # Pero sanitizar el contenido
        comment = response.json()
        assert '<script>' not in comment['content']
        assert '<b>Bold text</b>' in comment['content']  # HTML seguro debe mantenerse
```

5. Herramientas de Security Testing (5 min)

Análisis estático de seguridad:

```bash
# Bandit - Encuentra vulnerabilidades comunes en Python
bandit -r src/ -f json -o security_report.json

# Safety - Verifica dependencias con vulnerabilidades conocidas
safety check --json --output security_deps.json

# Semgrep - Análisis avanzado de código
semgrep --config=auto src/
```

Testing dinámico:

```bash
# OWASP ZAP - Automated security testing
zap-cli quick-scan http://localhost:5000

# Nuclei - Fast vulnerability scanner
nuclei -u http://localhost:5000 -t vulnerabilities/

# SQLMap - Automated SQL injection testing
sqlmap -u "http://localhost:5000/search?q=test" --batch
```

Testing de APIs:

```python
# Utilizando pytest con requests
def test_api_security_headers():
    response = requests.get('http://localhost:5000/api/users')

    # Verificar headers de seguridad
    assert 'X-Content-Type-Options' in response.headers
    assert response.headers['X-Content-Type-Options'] == 'nosniff'
    assert 'X-Frame-Options' in response.headers
    assert 'X-XSS-Protection' in response.headers
```

Fuzzing automatizado:

```python
# Usando hypothesis para fuzzing
from hypothesis import given, strategies as st

@given(st.text(min_size=1, max_size=1000))
def test_search_endpoint_fuzzing(random_input):
    """Fuzzing del endpoint de búsqueda"""
    response = client.get(f'/search?q={random_input}')
    # No debe crashear nunca
    assert response.status_code in [200, 400, 422]
    # No debe revelar información sensible
    assert 'password' not in response.text.lower()
    assert 'secret' not in response.text.lower()
```

6. Automatización de Security Testing (5 min)

Pipeline de seguridad integrado:

yaml

```yaml
# .github/workflows/security.yml
name: Security Testing
on: [push, pull_request]

jobs:
  security-scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Static Security Analysis
        run: |
          pip install bandit safety
          bandit -r src/ -f json -o bandit-report.json
          safety check --json --output safety-report.json

      - name: Security Unit Tests
        run: |
          pytest tests/security/ -v --junitxml=security-tests.xml

      - name: Dynamic Security Testing
        run: |
          # Start application
          python app.py &
          sleep 10

          # Run security tests
          zap-cli quick-scan http://localhost:5000

      - name: Upload Security Reports
        uses: actions/upload-artifact@v3
        with:
          name: security-reports
          path: |
            bandit-report.json
            safety-report.json
            security-tests.xml
```

Continuous Security Monitoring:

- **Dependency scanning** → Automated updates for vulnerable packages
- **Secret scanning** → Detect hardcoded credentials
- **License compliance** → Verify open source licenses
- **Container security** → Scan Docker images for vulnerabilities

---

🛡️ Security Testing Checklist

Authentication & Session:

- ✅ Password complexity requirements
- ✅ Account lockout after failed attempts
- ✅ Session timeout implementation
- ✅ Secure cookie configuration
- ✅ Token expiration and refresh
- ✅ Multi-factor authentication (if applicable)

Authorization & Access Control:

- ✅ Role-based access enforcement
- ✅ Principle of least privilege
- ✅ Direct object reference protection
- ✅ Admin function isolation
- ✅ API endpoint protection

Input Validation:

- ✅ All inputs validated server-side
- ✅ SQL injection prevention
- ✅ XSS protection
- ✅ Path traversal prevention
- ✅ File upload security
- ✅ JSON/XML parsing security

Data Protection:

- ✅ Sensitive data encryption

- ✅ Secure password storage
- ✅ Data exposure prevention
- ✅ Audit logging implementation

⚠️ Common Security Testing Pitfalls

❌ Mistakes to avoid:

- **Testing only happy paths** → Attackers use malicious inputs
- **Trusting client-side validation** → Always validate on server
- **Not testing edge cases** → Boundary conditions often vulnerable
- **Ignoring error messages** → They can leak sensitive information
- **Focusing only on code** → Configuration and deployment matter

✅ Best practices:

- **Assume malicious users** → Test with attacker mindset
- **Validate everything** → Every input, every parameter
- **Test negative scenarios** → What happens when things go wrong?
- **Use realistic test data** → Production-like data reveals more issues
- **Automate security testing** → Manual testing misses things

---

📝 Resumen Final

1. **Security testing** requiere mentalidad diferente a functional testing
2. **OWASP Top 10** proporciona framework para priorizar vulnerabilidades
3. **Authentication/Authorization** son áreas críticas para testing exhaustivo
4. **Input validation** debe testear todos los vectors de ataque posibles
5. **Herramientas automatizadas** complementan pero no reemplazan testing manual
6. **Pipeline integration** hace security testing parte del proceso de desarrollo
7. **Continuous monitoring** detecta nuevas vulnerabilidades en dependencias

---

🏃 Actividad Práctica Sugerida

Ejercicio 1 - Vulnerability Testing:

1. Implementar endpoint vulnerable a SQL injection

2. Crear test que demuestre la vulnerabilidad
3. Arreglar la vulnerabilidad
4. Validar que el test ahora pase

Ejercicio 2 - Authentication Testing:

1. Crear suite de tests para login/logout
2. Implementar tests de brute force protection
3. Validar session management security
4. Test de privilege escalation prevention

Tiempo estimado: 30-35 minutos adicionales