Homework 3: Web Security

Computer Security

Universidad San Francisco de Quito

Xavier Sebastián Tandazo Cobo

2025

1. Task A: Categorization and Description

Según Stallings (Capítulo 21), un ataque Distributed Denial of Service (DDoS) intenta consumir recursos críticos del sistema víctima, ya sea **recursos internos del host** (o estructuras como tablas TCP) o **capacidad de transmisión en la red**. Esta clasificación coincide con las categorías modernas usadas hoy en ciberseguridad.

A continuación se presentan las tres categorías fundamentales y dos ejemplos para cada una.

1. Volumetric Attacks

Los ataques volumétricos buscan saturar por completo la capacidad de transmisión de datos de la red víctima. Como explica Stallings (Capítulo 21), este tipo de ataque consume el *data transmission capacity*, impidiendo que tráfico legítimo pueda circular.

> Supongamos que una pequeña empresa utiliza un enlace de Internet de 500 Mbps para operar su sitio web. Un atacante coordina una botnet de miles de equipos infectados repartidos por distintos países. A una hora específica, el botnet comienza a enviar tráfico masivo hacia los servidores de la empresa, es decir, mucho más del ancho de banda disponible, produciendo una congestión total. Aunque el servidor de la empresa siga encendido y funcional, su canal de comunicación queda saturado y los clientes no pueden acceder al servicio.

Entre los ataques volumétricos más comunes (descritos también por Stallings) se encuentran:

- **UDP Flood:** Envío masivo de datagramas UDP hacia puertos aleatorios para saturar el ancho de banda.
- **Smurf Attack:** Los atacantes envían solicitudes ICMP a una red intermediaria con la dirección IP de la víctima, provocando que muchos dispositivos respondan simultáneamente al objetivo.

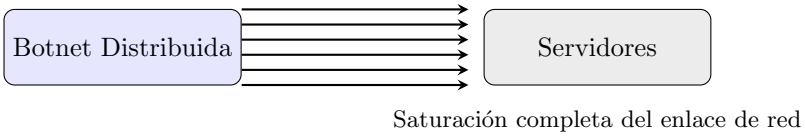


Figure 1: Ejemplo de ataque volumétrico: tráfico masivo generando congestión total.

2. Protocol-Based Attacks

Los ataques basados en protocolo no buscan saturar el ancho de banda, sino agotar recursos internos del sistema víctima, como tablas TCP, buffers de conexión o estructuras de seguimiento de estado.

Según Stallings (Capítulo 21), estos ataques explotan el funcionamiento normal de los protocolos para forzar al servidor a mantener estados incompletos o inválidos.

> Supongamos que una entidad financiera administra un servidor capaz de manejar hasta 20,000 conexiones TCP activas. Un atacante despliega cientos de bots que comienzan a enviar miles de paquetes TCP `SYN` con direcciones IP de origen falsificadas. Cada petición aparenta iniciar una conexión legítima, pero nunca completará el *three-way handshake*.
>
> El servidor, siguiendo el protocolo TCP, responde a cada solicitud con un `SYN/ACK` y crea una entrada temporal en su tabla TCP, reservando recursos mientras espera un `ACK` que nunca llegará. En pocos segundos, la tabla se colma de estas conexiones incompletas o *half-open*. Como resultado, las conexiones legítimas ya no pueden registrarse en la tabla y el servicio se vuelve inaccesible, aun cuando el ancho de banda no está saturado.

Ejemplos comunes de ataques basados en protocolo serían:

- **SYN Flood:** El atacante envía miles de solicitudes TCP SYN sin completar el handshake, llenando la tabla de estados.
- **Ping of Death:** Se envía de paquetes ICMP malformados o sobredimensionados que pueden provocar errores internos o reinicios.

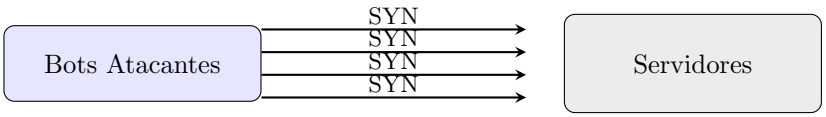


Tabla TCP saturada con conexiones half-open

Figure 2: Ataque basado en protocolo: agotamiento de estructuras internas del servidor.

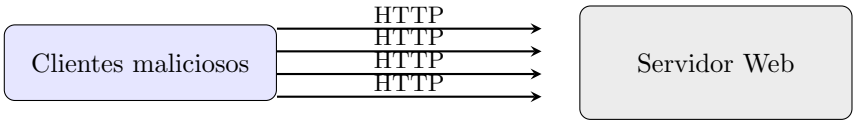
3. Application Layer Attacks (L7)

Los ataques de capa de aplicación (L7) apuntan directamente a los recursos lógicos de la aplicación, como hilos HTTP, ciclos de CPU o consultas a bases de datos. A diferencia de los ataques volumétricos o basados en protocolo, el tráfico suele parecer legítimo, lo que hace más difícil distinguir al atacante de un usuario real.

> Pensemos en una empresa cuyo servidor web procesa miles de peticiones legítimas cada minuto. Un atacante organiza un conjunto de clientes maliciosos que comienzan a enviar solicitudes `HTTP GET` muy costosas —por ejemplo, páginas que requieren múltiples consultas a la base de datos o que generan reportes dinámicos.
>
> Aunque cada petición parece normal, la cantidad acumulada satura los hilos de ejecución del servidor web. Al poco tiempo, todas las conexiones disponibles están ocupadas atendiendo solicitudes falsas, impidiendo que los clientes reales accedan a la página. Pero el servidor no está caído; simplemente está demasiado ocupado atendiendo tráfico aparentemente válido.

Ejemplos comunes incluyen:

- **HTTP GET/POST Flood:** Solicitudes masivas que activan operaciones costosas.
- **Slowloris:** Conexiones HTTP que se mantienen abiertas enviando encabezados incompletos, agotando los hilos del servidor.



Hilos de aplicación ocupados por solicitudes costosas

Figure 3: Ataque de capa de aplicación: agotamiento de recursos lógicos del servidor web.

Categoría	OSI	Objetivo Principal	Ejemplo 1	Ejemplo 2
Volumétricos	L3–L4	Saturar la **capacidad de transmisión** del enlace para impedir que el tráfico legítimo llegue al destino.	UDP Flood	Smurf Attack
Protocol-Based	L3–L4	Agotar **recursos internos del host** explotando el funcionamiento normal de los protocolos (tablas TCP, buffers, estados).	SYN Flood	Ping of Death
Application Layer (L7)	L7	Consumir **recursos lógicos de la aplicación** (hilos HTTP, CPU, consultas a BD) mediante tráfico que aparenta ser legítimo.	HTTP GET/POST Flood	Slowloris

Table 1: Comparación de las tres categorías principales de ataques DDoS según su objetivo, capa OSI y ejemplos representativos.

1. Task B: In-Depth Attack Profile (HTTP Flood)

Como dijimos, según Stallings (Capítulo 21), los ataques DDoS modernos no dependen únicamente de tráfico volumétrico, sino también de técnicas que explotan las limitaciones lógicas de servicios específicos.

El ejemplo representativo, que hemos utilizado, es el **HTTP Flood**, un ataque de capa de aplicación (L7) en el que miles de solicitudes HTTP aparentemente legítimas saturan los recursos del servidor web, aun cuando el ancho de banda total no se aproxima a su límite.

Mecanismo

El ataque consiste en que el adversario dirige grandes volúmenes de solicitudes `HTTP GET` o `POST` hacia páginas que requieren procesamiento intensivo. Cada petición es válida, cumple el estándar HTTP y no presenta características anómalas a nivel de red. Por lo tanto, coincidiendo con la descripción de Stallings, este ataque se vuelve difícil de distinguir del tráfico real sin afectar la disponibilidad para usuarios legítimos.

A diferencia de ataques reflectores como DNS Amplification, el HTTP Flood no depende del tamaño del paquete, sino del *costo computacional* que implica generar una respuesta.

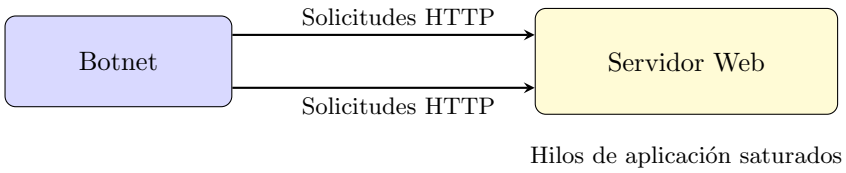


Figure 4: Mecanismo del ataque HTTP Flood.

Agotamiento de recursos

El recurso agotado no es el ancho de banda, sino la **capacidad de procesamiento del servidor**. Cada solicitud activa hilos del *application pool*, consume CPU y puede generar consultas a la base de datos. Cuando todos los hilos se ocupan procesando peticiones falsas, el servidor permanece en línea, pero deja de responder a clientes reales.

Stallings destaca que los ataques de capa de aplicación pueden ser altamente efectivos sin necesidad de tráfico masivo, justamente porque el costo de procesamiento recae desproporcionadamente sobre la víctima.

Botnets y Amplificación

Tal como describe Stallings, los atacantes emplean botnets construidas a través de técnicas de escaneo como *random scanning*, *hit-list scanning* o *topological scanning*. En un HTTP Flood, la botnet distribuye las solicitudes entre miles de máquinas, simulando usuarios reales y dificultando la aplicación de filtros basados en IP o en volumen.

Aunque el ataque no utiliza amplificación en el sentido tradicional (como DNS o NTP), sí se beneficia de una **amplificación a nivel de aplicación**: una sola solicitud HTTP puede desencadenar operaciones costosas, como consultas SQL, generación dinámica de contenido o lectura de archivos grandes.

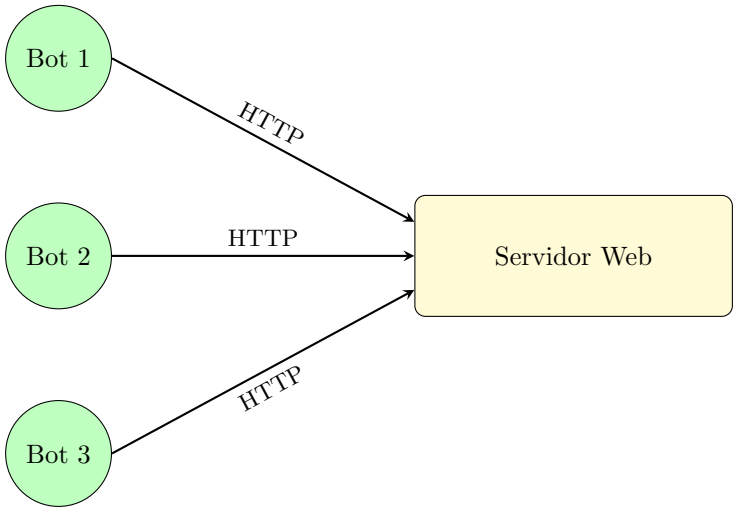


Figure 5: Botnets enviando tráfico HTTP malicioso pero aparentemente legítimo hacia el servidor.

2. Task A: DVWA Installation via Docker

Para el trabajo práctico se desplegó localmente la aplicación "Damn Vulnerable Web Application" (DVWA) utilizando Docker, siguiendo las instrucciones del enunciado. El contenedor se inició con el siguiente comando:

```
docker run --rm -it -p 80:80 vulnerables/web-dvwa
```

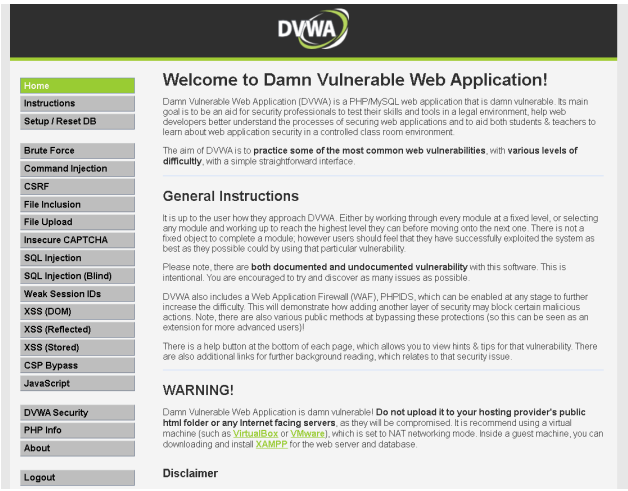


Figure 6: Página Inicial de DVWA una vez se inicia sesión.

Tras acceder a `http://localhost/`, iniciar sesión y completar la inicialización de la base de datos, se procedió a cambiar el nivel de seguridad a **Medium**.

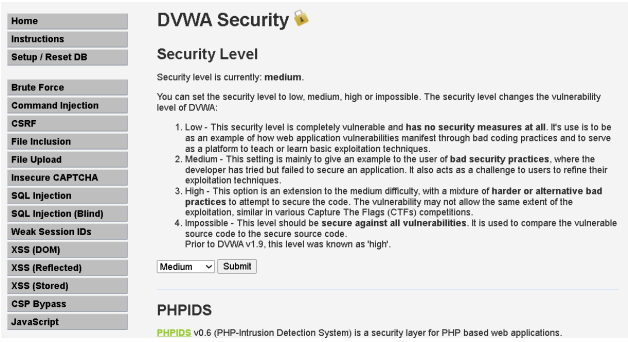


Figure 7: Página de seguridad de DVWA donde se modifica el nivel a Medium

2. Task B: Execution of the attack in DVWA

SQL Injection (Medium Security)

Vulnerability: SQL Injection (SQLi)

En el nivel Medium, DVWA aplica un filtrado básico que escapa comillas mediante funciones como `mysql_real_escape_string()`, e impide ingresar texto libre reemplazando el campo de entrada por un menú desplegable con valores limitados del 1 al 5. Esto bloquea los payloads simples utilizados en el nivel Low.

Para analizar la solicitud enviada por el formulario, utilizamos la herramienta de desarrollador (F12) y la pestaña *Network*. Allí observamos la petición exacta enviada al servidor al presionar *Submit*.

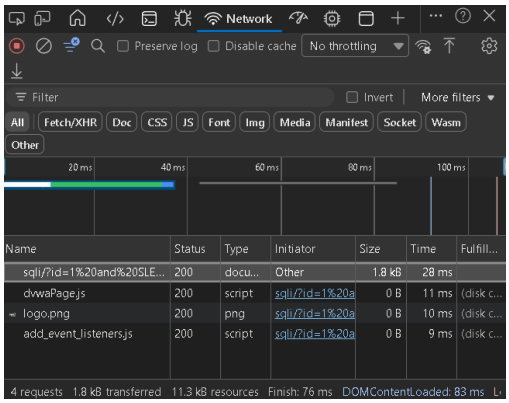


Figure 8: Request capturada al enviar el formulario.

Copiamos esta solicitud como un comando `curl`, lo que permitió modificar el parámetro `id` manualmente. Primero probamos un payload típico con comillas:

```
id=1' or 1=1#
```

El servidor devolvió un error de sintaxis SQL:

```
You have an error in your SQL syntax near '\' or 1=1#'
```

Esto indicó que el filtro escapaba las comillas, invalidando el payload. A continuación probamos una versión sin comillas:

Payload utilizado:

```
id=1 or 1=1#
```

Esta variante sí funcionó, y el servidor devolvió todos los usuarios en la tabla en lugar de un solo registro:

```
<div class="body_padded">
        <h1>Vulnerability: SQL Injection</h1>



        <div class="vulnerable_code_area">
                <form action="#" method="POST">
                        <p>
                                User ID:
                                <select name="id"><option value="1">1</option><option value="2">2</option><option va
lue="3">3</option><option value="4">4</option><option value="5">5</option></select>
                                <input type="submit" name="Submit" value="Submit">
                        </p>

                </form>
                <pre>ID: 1 or 1=1#<br />First name: admin<br />Surname: admin</pre><pre>ID: 1 or 1=1#<br />First nam
e: Gordon<br />Surname: Brown</pre><pre>ID: 1 or 1=1#<br />First name: Hack<br />Surname: Me</pre><pre>ID: 1 or 1=1#
<br />First name: Pablo<br />Surname: Picasso</pre><pre>ID: 1 or 1=1#<br />First name: Bob<br />Surname: Smith</pre>
        </div>
```

Figure 9: Explotación exitosa del SQLi utilizando cURL.

El nivel Medium solo filtra comillas, pero el backend no envuelve el parámetro entre comillas dentro de la consulta SQL. Esto permite inyectar operadores lógicos como `OR 1=1` directamente, evitando la protección.

Bypass alternativo: modificación del HTML del formulario

Como segunda técnica, evitamos la restricción del menú desplegable editando el HTML del formulario mediante la herramienta de desarrollador. Modificamos el valor del primer elemento de la lista para reemplazarlo por el payload:

```
<option value="1 or 1=1#">1</option>
```

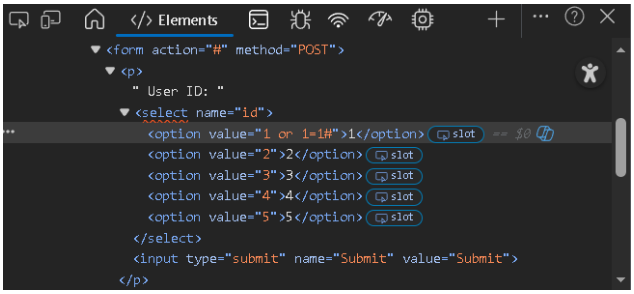


Figure 10: Modificación del HTML del formulario para incluir el payload.

Al enviar el formulario, el navegador transmitió el valor manipulado y el backend ejecutó la inyección correctamente. Nuevamente se obtuvieron todos los registros de la tabla:

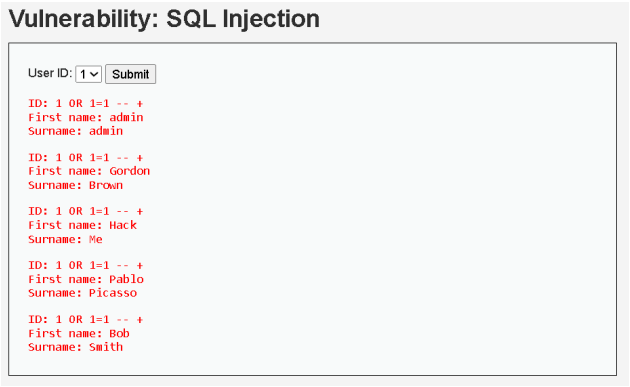


Figure 11: Resultado del SQLi tras modificar el HTML del formulario.

Un caso típico en sistemas de inicio de sesión mal protegidos (realizado por principiantes en su mayoría) podría ser el siguiente. Un atacante podría escribir:

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

Si el sistema concatena cadenas sin validación, podría terminar ejecutando:

```
SELECT * FROM usuarios WHERE usuario='' OR '1'='1' AND password=''
```

lo que permitiría iniciar sesión sin credenciales válidas, comprometiendo cuentas administrativas o datos sensibles.

Cross-Site Scripting (Reflected XSS) — Medium Security

Vulnerability: Cross-Site Scripting (Reflected XSS)

En el módulo XSS con DVWA en nivel **Medium** se asumió que el sistema debía implementar un filtrado parcial contra XSS, es decir, que el servidor busca específicamente la cadena literal `<script>` en minúsculas y la elimina o neutraliza antes de reflejar la entrada al usuario. Todo esto ya que en el nivel **Low** se comprobó que no tiene este filtrado, además, la estructura de la pregunta y el campo de la respuesta le deja al usuario total llibertad de escribir lo que desee, entonces la seguridad tenía que ir por ese lado.

Para comprobar este comportamiento, primero se probó un payload clásico para mostrar una alerta (uno que previamente funcionó en el nivel **Low**):

```
<script>alert('XSS')</script>
```

El resultado fue el que se esperaba, la etiqueta `<script>` fue filtrada por completo, de manera que el navegador no ejecutó el código JavaScript. En cambio, la aplicación simplemente trató el contenido entre etiquetas como texto, reflejándolo en pantalla sin producir ninguna alerta.

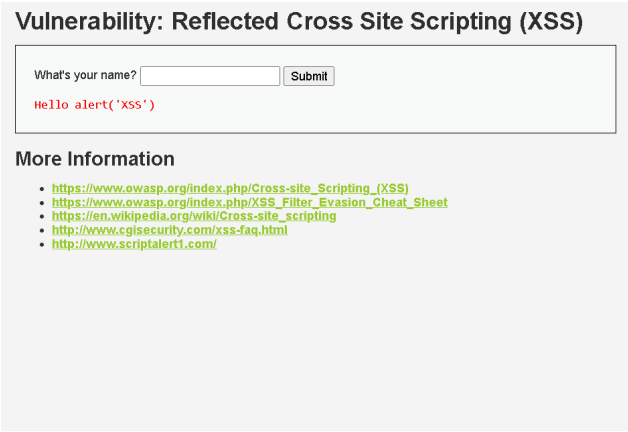


Figure 12: Resultado sin alertas: el servidor filtra `<script>` y muestra "alert('XSS')" como texto.

A partir de este comportamiento se concluyó que el sistema no realiza un filtrado completa ni un análisis estructural del HTML, sino únicamente un reemplazo textual muy limitado. En particular, el nivel **Medium** no normaliza mayúsculas o minúsculas, por ejemplo, ni mucho menos detectaría variaciones más complejas de la etiqueta.

Por tanto, se planteó la hipótesis de que un payload que fragmentara la cadena `<script>` o modificara su estructura impediría al filtro detectarla, pero que aun así el navegador reconstruiría la etiqueta válida durante el parseo del HTML, ejecutando el código JavaScript incrustado. Se toma en cuenta lo siguiente:

- **Los filtros débiles actúan a nivel de texto, no a nivel de DOM.** Es decir, tal y como aprendimos en el nivel **Low** de SQL Injection, el servidor solo revisa el string literal, en este caso sería `<script>`. Si este se altera mínimamente, el filtro ya no lo reconoce.
- **El navegador sí interpreta el HTML de forma flexible.** Por eso, aunque la etiqueta llegue fragmentada o interrumpida, el navegador puede recomponerla y ejecutarla si el resultado final conforma una etiqueta válida.

Con esta lógica, se construyó un payload que dividiera el token `script` en varias partes, intercalando una etiqueta `<script>` en medio para confundir el filtro pero permitir que el navegador reconstruyera la estructura final:

Payload usado (registro para la entrega):

```
<scr<script>ipt>alert('XSS')</scr<script>ipt>
```

La ejecución de la alerta confirma que el navegador logró recomponer una etiqueta `<script>` válida a partir de los fragmentos, demostrando que el filtro del nivel Medium no es capaz de detectar variantes no literales de la cadena prohibida. Esto constituye una explotación exitosa de XSS reflejado.

Posteriormente, la respuesta HTML mostró únicamente el texto "Hello" sin incluir el nombre enviado en el formulario.

Un ejemplo común ocurriría en formularios de búsqueda. Si la aplicación refleja el parámetro sin validarlo, un atacante podría enviar a un usuario un enlace como:

```
https://pagina-web-prueba.com/buscar?q=<script>document.cookie</script>
```

El navegador del usuario ejecutará el código, permitiendo al atacante robar cookies de sesión, modificar el contenido de la página o redirigir al usuario a sitios maliciosos.

Figure 13: Alerta desplegada por la ejecución del payload.

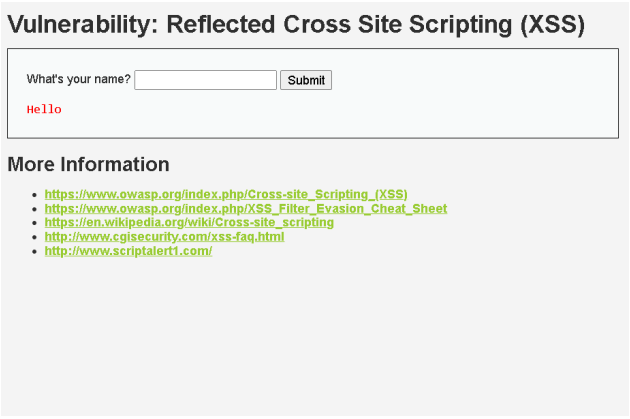


Figure 14: Respuesta de la página mostrando "Hello" sin el nombre; el payload fue ejecutado antes del renderizado.

Command Injection — Medium Security

Vulnerability: Command Injection (Argument Injection)

En el nivel **Medium**, DVWA ya no permite la ejecución de comandos adicionales como en el nivel Low. Asumimos que el backend aplica la función `escapeshellcmd()`, la cual elimina o neutraliza caracteres peligrosos tales como `; | & ‘`, `$`, >, < y otros operadores que normalmente se utilizan para encadenar comandos. Debido a esto, payloads típicos como:

```
8.8.8.8; ls -la
8.8.8.8 && ls -la
8.8.8.8 || ls -la
```

son filtrados por completo y el sistema únicamente ejecuta el comando permitido (`ping`) sin aceptar instrucciones adicionales. Durante las pruebas confirmamos que ninguno de estos métodos produjo salida distinta de la del `ping` normal, lo cual indica que el servidor bloquea efectivamente la inyección de comandos externos.

Al revisar el comportamiento del sistema, se concluyó que aunque el servidor elimina operadores de concatenación, **no valida los parámetros que pertenecen al comando permitido**. Esto abre la puerta a un segundo vector: la inyección de argumentos directamente dentro del comando `ping`. Es decir, aunque no podemos ejecutar nuevos comandos, sí podemos manipular el comportamiento del comando existente.

Para comprobarlo, se probaron diferentes banderas válidas de `ping`, incluyendo parámetros que modifican el tamaño del paquete, la cantidad de fragmentación o el número de repeticiones. En este caso no centraremos en el siguienete ejemplo:

Payload utilizado:

```
8.8.8.8 -s 65000
```

El parámetro `-s` indica el tamaño del paquete ICMP, y un valor tan alto como `65000` obliga al servidor a procesar una solicitud excesivamente grande. El resultado fue un retraso notable en la respuesta del servidor, tardando varios segundos antes de mostrar la salida del `ping`. En algunos casos, dependiendo de la configuración del entorno, esto puede incluso generar un efecto similar a un *Denial of Service* local por consumo de recursos.

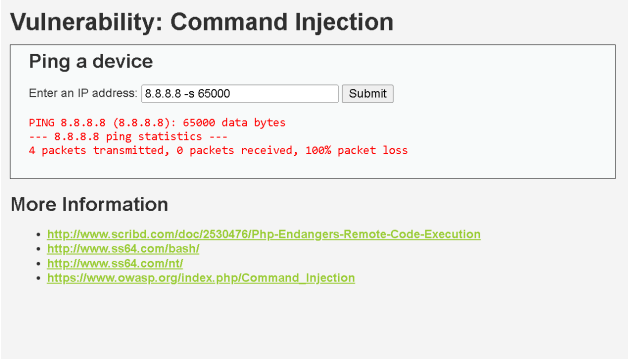


Figure 15: Resultado del payload, renderizado después de 10 segundos (varía según el hardware del usuario)

Este resultado demuestra que, aunque DVWA Medium bloquea la inyección de comandos adicionales, el filtro sigue siendo insuficiente porque no verifica si la entrada del usuario introduce argumentos nuevos dentro del comando legítimo. En consecuencia, es posible manipular completamente la ejecución del `ping` sin necesidad de utilizar operadores prohibidos.

Durante las pruebas también se identificó una vulnerabilidad adicional no documentada inicialmente. Aunque `escapeshellcmd()` bloquea operadores de concatenación como , se observó que la función de filtrado no actúa correctamente cuando dichos caracteres aparecen en combinación con comillas. Esto permitió ejecutar un payload como:

```
8.8.8.8 & echo "Te voy a Hackear"
```

El servidor procesó la segunda instrucción, mostrando el mensaje `Te voy a Hackear` en la salida del comando. Este comportamiento confirma que el filtrado implementado no sanitiza adecuadamente los argumentos cuando incluyen comillas, lo que permite evadir parcialmente la protección y ejecutar comandos adicionales pese a las restricciones del nivel.

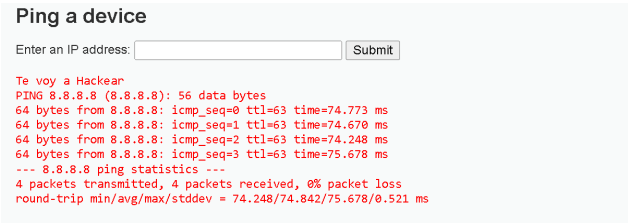


Figure 16: Ejecución del comando inyectado, mostrando el mensaje `"Te voy a Hackear"` en la salida.

Un escenario común a los ejemplos que podría suceder en la vida real podría suceder si una aplicación web permite al usuario ejecutar operaciones de diagnóstico, como un `ping`, `traceroute` o `nslookup`, sin validar los parámetros que el usuario puede añadir. Por ejemplo, en muchos paneles de administración de routers, firewalls o dispositivos IoT existe un formulario similar a:

```
ping <X.X.X.usuario>
```

Si el sistema no valida los argumentos, un atacante podría enviar:

```
google.com -s 65000
```

Esto obliga al dispositivo a enviar paquetes ICMP extremadamente grandes, consumiendo ancho de banda, memoria y CPU.

3. Task C: Defensive Installation: Web Application Firewall (WAF)

Para esta sección, se seleccionó **ModSecurity** como WAF, ejecutándose sobre un contenedor **Nginx** configurado como *reverse proxy* delante del contenedor de DVWA.

Montaje de los contenedores

El despliegue se realizó utilizando **Docker Compose**, levantando tres contenedores principales:

- **DVWA:** Aplicación vulnerable con la base de datos conectada.
- **MariaDB:** Base de datos para DVWA.
- **Nginx + ModSecurity:** WAF que actúa como reverse proxy y aplica reglas CRS.

Todo el tráfico HTTP se dirigió a través del WAF hacia DVWA, manteniendo las sesiones de usuario y encabezados intactos. Se verificó primero en modo `DetectionOnly` y luego se activó `Enforce` para bloquear ataques.

C: > Users > xavie > Desktop > waf > docker-compose.yml

```yaml
services:
  dvwa:
    image: vulnerables/web-dvwa
    container_name: dvwa
    environment:
      - MYSQL_USER=dvwa
      - MYSQL_PASSWORD=p4ssw0rd
      - MYSQL_DATABASE=dvwa
      - MYSQL_HOST=dvwa-db
    ports:
      - "8080:80"
    depends_on:
      - dvwa-db
    networks:
      - dvwa-net
  ▷Run Service
  dvwa-db:
    image: mariadb:10.11
    container_name: dvwa-db
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=dvwa
      - MYSQL_USER=dvwa
      - MYSQL_PASSWORD=p4ssw0rd
    networks:
      - dvwa-net
  ▷Run Service
  waf:
    image: owasp/modsecurity-crs:nginx
    container_name: waf
    ports:
      - "8000:80"
    volumes:
      - ./modsec.conf:/etc/nginx/modsecurity.d/include.conf
    depends_on:
      - dvwa
    networks:
      - dvwa-net
```

Figure 17: Archivo `docker-compose.yml` utilizado para desplegar DVWA, base de datos y WAF.

Diagrama de red

Pruebas de defensa

Se realizaron las mismas pruebas de la Parte 2, con los siguientes resultados:

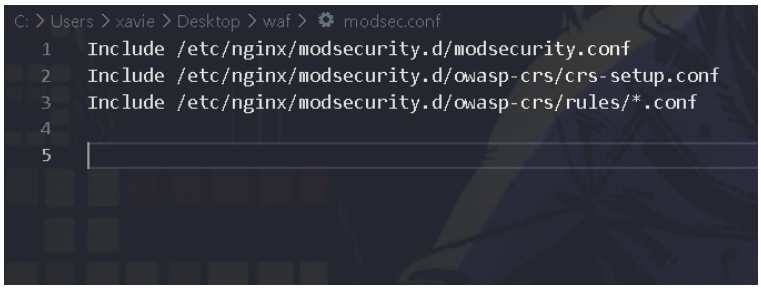


Figure 18: Configuración de ModSecurity con OWASP CRS sobre Nginx.

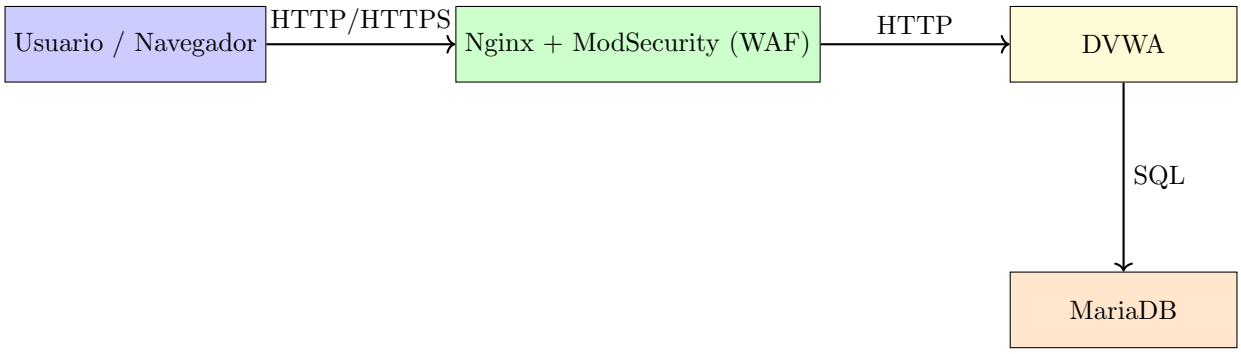


Figure 19: Diagrama de red mostrando el flujo de tráfico a través del WAF hacia DVWA y la base de datos.

- **SQL Injection:** El payload `id=1 OR 1=1#` fue interceptado por ModSecurity y bloqueado con un `HTTP 403 Forbidden`. La regla CRS detectó el patrón malicioso antes de que llegara al backend.

Figure 20: Bloqueo de un SQLi por ModSecurity, mostrando HTTP 403.

- **Cross-Site Scripting (XSS):** Los payloads que incluían `<script>` fueron neutralizado por el WAF, evitando la ejecución de código en el navegador.
- **Command Injection:** Caracteres y patrones típicos de ejecución de comandos (`;`, `|`, `&&`) fueron bloqueados, aunque argumentos válidos del comando permitido no fueron interceptados. Esto demuestra la necesidad de validación adicional en el backend.
- **HTTP Flood simulado:** El WAF puede limitar solicitudes concurrentes y aplicar reglas de *rate-limiting* básicas, reduciendo parcialmente el impacto de ataques volumétricos a nivel de aplicación.

Análisis y comparación de ataques

El WAF demostró eficacia frente a ataques de **capa de aplicación (L7)** como SQLi y XSS, bloqueando solicitudes con patrones maliciosos y devolviendo HTTP 403. Por el contrario, ataques como HTTP Flood buscan saturar recursos del servidor mediante tráfico legítimo distribuido, algo que un WAF estándar solo puede mitigar parcialmente.

Esto evidencia que:

- Los ataques dirigidos a la lógica de la aplicación requieren inspección de contenido y reglas específicas; un WAF bien configurado los mitiga efectivamente.
- Los ataques volumétricos o basados en protocolo necesitan medidas adicionales a nivel de red para proteger la disponibilidad.
- La combinación de un WAF con validación robusta del backend, monitoreo de tráfico y control de recursos proporciona una defensa integral frente a amenazas L7.

El despliegue de ModSecurity sobre Nginx demostró ser una barrera efectiva frente a inyecciones y ataques de aplicación, aunque la protección completa contra ciertos vectores requiere controles complementarios en la capa de aplicación y de red.