Team Name: Security-Sonnets
CSE 418/518 Software Security
Course Project Deliverable 2-2

WebDenial

Adam Xu (50495452)
John Chen (50378336)
Junhao Chen (50414682)
Shusen ZHENG (50629157)

Overview:

The WebDenial project is a security-focused website designed to find and fix weaknesses in modern web systems. The project focuses on building secure features and testing the site against attacks, especially Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks.

Project Goals:

Basic Website Features:

- Build essential functions like user login and registration, role-based access (different permissions for different users), basic operations (create, read, update, delete), a solid database setup, and a system for notifications. These features make the site work well while keeping it secure.

Better Security Measures:

- Use techniques like encrypting data, preventing HTML injection attacks, and using token-based checks. These steps help protect sensitive information and protect data from common web attacks.

DoS/DDoS Research and Testing:

- Learn about and document different DoS/DDoS attacks, such as SYN Flood, HTTP Flood, Slowloris, and Amplification Attacks. The project will simulate these attacks to see how they affect the website and to check if our security measures work.

How We Will Work:

Development and Testing:

- Start by designing and building a basic version of the site with security features already in place. Then, test the site thoroughly by simulating attacks and measuring how well it performs under stress.

Defense Implementation:

- After testing, extra methods will be added to fix any weaknesses found. This includes techniques like limiting the rate of requests, blocking suspicious IP addresses, using CAPTCHAs, and other measures to ensure the site stays available even during an attack.

Functions:

1. **User Authentication System**
    - The system employs secure password storage using bcrypt, token-based session management, and HTTP-only cookies to safeguard user sessions, ensuring that only valid, authenticated users can access protected routes.
2. **Role-Based Access Control**
    - System restricts access to the ability of the user based on the user's role. Privileged account "Admin" should have authority to delete all user's posts while regular users can only delete posts created by them.
3. **Basic CRUD Operations**
    - The system manages posts by storing and retrieving data from the database. Posts are fetched by the database which allows their post to be public. Regular users are only allowed to delete their own post.

4. **Database Design**
    - The design leverages MongoDB with separate collections for credentials and posts, enabling flexible document storage and retrieval while emphasizing the importance of security and efficient indexing for scalability.
5. **Notification System**
    - User will be notify by email when they successfully register an account. Similarly, when a user deletes their account, a notification email will be sent to the email confirming the deletion and inform them to enhance account security,

Security:

1. **Data Encryption**

    Register functions bcrypt to securely hash passwords before storing them in the database. The function bcrypt.hashpw() generates a salted hash, making it computationally expensive to reverse. This enhances security by protecting passwords from brute-force and rainbow table attacks.

2. **HTML Injection:**

    Flask's render_template function escapes content by default, which helps mitigate basic HTML injection

3. ***Auth_Token:***

    The application generates a secure random token, hashes it using SHA-256, and stores the hash in the database while setting the raw token in an HTTP-only cookie. This token is then validated on each request to ensure session integrity.

4. **DoS/DDoS Defense Mechanisms:**

    Refer to the **DoS/DDoS Research** section below.

DoS/DDoS Research:

1. Definition of DoS/DDoS Attacks:

   Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks are malicious attempts to disrupt the regular traffic of a targeted server, service, or network by overwhelming it with a flood of Internet traffic. These attacks exploit various network and application-level vulnerabilities.

2. Common types of DoS/DDoS attacks:

   a. **SYN Flood Attack**
      i. Mechanism: Exploits the TCP handshake process by sending numerous SYN (synchronized) requests to a target server but never completing the handshake.
      ii. Impact: The server keeps waiting for the handshake completion, exhausting its resources and making it unable to accept legitimate connections.
   b. **HTTP Flood Attack**
      i. Mechanism: Overwhelms a web server by sending many HTTP requests, mimicking legitimate users but at an extremely high rate.
      ii. Impact: Server resources are consumed, leading to slow response times or unavailability.
   c. **Slowloris Attack**
      i. Mechanism: Sends partial HTTP requests to a server and keeps connections open as long as possible by sending small, slow data packets.
      ii. Impact: The server's connection pool gets exhausted, preventing new users from accessing the service.
   d. **Amplification Attacks**
      i. Mechanism: Attackers use spoofed requests to send small queries to publicly accessible services like DNS, NTP, or Memcached, which then send disproportionately significant responses to the victim.
      ii. Impact: The target network is flooded with large amounts of data, consuming its bandwidth.

3. Implementation of DoS/DDoS Attacks

This project uses a Locust-based test to simulate a Denial-of-Service (DoS) scenario by rapidly sending multiple requests to your local server (http://localhost:8080). The details are shown below step by step:

a. **Simulating a DoS Attack with Locust**

Firstly, the locust spawns multiple virtual users (--users 500), each repeatedly sending requests to the server. Then, the users are added gradually (--spawn-rate 100), simulating an increasing load. Each user sends a GET request to /over a period (--run-time 5m). Finally, the locust records the response times, request success/failure rates, and server throughput.

```python
from locust import HttpUser, task, between

class StressTestUser(HttpUser):
    wait_time = between(0.1, 0.5)  # Simulates rapid-fire requests

    @task
    def flood_server(self):
        """Simulate a DoS attack with repeated requests"""
        self.client.get("/")  # Modify if your API uses different endpoints
```

b. **Collecting Performance Data**

After the DoS Attack is finished, the locust logs the results in CSV files:

```
# Run with report summarizing:
#   locust -f locustfile.py --host http://localhost:8080 --headless
#       --users 500 --spawn-rate 100 --run-time 5m --csv=locust_results
```

This will create files: “locust_results_stats.csv” (stores request counts, response times, and RPS.) and “locust_results_failures.csv” (stores failed requests and errors.)

c. **Analyzing the Server's Response**

Another script, analyze_locust.py, will then read the Locust logs using Pandas, after which it calculates the Total Requests Sent, Average Response Time (ms), Requests Per Second (RPS), and Total Failures &

Failure Rate (%). The report is shown below:

```
# ====== Load Test Report ======
# Total Requests: 25000
# Average Response Time: 230.45 ms
# Requests Per Second (RPS): 98.6
# Total Failures: 1200
# Failure Rate: 4.8%
# ==============================
```

4. Defense Mechanisms

This project implements a basic rate-limiting mechanism using Redis to prevent excessive requests from a single IP address within a defined time window. The function rate_limit(ip, endpoint) checks whether an IP has exceeded the allowed number of requests (RATE_LIMIT = 5) within the specified duration (TIME_WINDOW = 60 seconds). If an IP has not reached the limit, the request count is incremented; otherwise, further requests are denied with a 429 Too Many Requests response.

```python
RATE_LIMIT = 5  # Max requests
TIME_WINDOW = 60  # Time window in seconds

def rate_limit(ip, endpoint):
    """ Check if the IP exceeds the request limit for the given endpoint """
    key = f"rate_limit:{endpoint}:{ip}"
    requests = rate_limiter.get(key)

    if requests is None:
        rate_limiter.set(key, 1, ex=TIME_WINDOW)
        return True
    elif int(requests) < RATE_LIMIT:
        rate_limiter.incr(key)
        return True
    else:
        return False
```

This approach effectively mitigates abuse and protects system resources by ensuring fair access. Potential future enhancements include granular rate limits, which are different endpoints that require different rate limits, and distributed rate limiting, which employing this mechanism across multiple servers with a shared Redis cluster would enhance scalability.

Summary:

In summary, the WebDenial project is a hands-on effort to build a secure web application that offers essential features like user log-in, role-based access, and data management and tests its resilience against common web attacks like DoS and DDoS. The project has already shown how a well-planned design and strong security measures can protect against real-world threats.

Looking Ahead

- **Expand Features:** Future project versions could add more advanced functions, such as enhanced user management and real-time monitoring dashboards, to better track and respond to potential security issues.

- **Improve Defenses:** There's potential to integrate smarter, adaptive defense mechanisms, including machine learning-based threat detection, which could automatically recognize and counteract new types of attacks.