Security Incident & Environment Hardening Report

Project: Thoutha (GraduationProject1)
Server Environment: Ubuntu Linux (1.9GB RAM, 4GB Swap)
Date: November 30, 2025

1. Threat Analysis: Observed Attacks

Analysis of the access.log file reveals that our server was immediately targeted by automated vulnerability scanners upon going public. This is standard behavior for any server connected to the internet; bots scan every public IP address continuously.

A. Attack Vectors Identified

1. **Directory Traversal Attacks**
   - **Method:** Attackers sent requests containing repeating /.%2e/ (URL-encoded ../) patterns.
   - **Target:** /cgi-bin/.%2e/.%2e/.../bin/sh
   - **Goal:** To break out of the web root folder and execute shell commands directly on our Linux server.
   - **Status:** **Blocked (400 Bad Request).** The embedded Tomcat server in Spring Boot correctly identified the malformed URL and rejected it before it reached our application logic.

2. **Sensitive File Probing**
   - **Method:** Automated scripts requested specific configuration files.
   - **Targets:** /.env (Environment variables), /.git/config (Source code version control), wp-login.php (WordPress), /swagger-ui.html (API docs).
   - **Goal:** To steal secrets (DB passwords, API keys) or find unpatched software.
   - **Status:** **Failed (404 Not Found).** our application does not expose these files at the root level.

3. **Vulnerability Scanners (Reconnaissance)**
    - **Actors:** IPs such as 206.189.225.181 and 167.99.61.240.
    - **User Agents:** l9scan, LeakIX, CensysInspect.
    - **Goal:** These are "indexing" bots that scan the entire internet to build databases of vulnerable servers. If they had found a vulnerability, our server would have been listed on public hacking search engines.

---

2. Applied Defenses & Mitigations

To protect the server against these threats while respecting its severe memory constraints (1.9GB RAM), we implemented a layered defense strategy.

Layer 1: Network Edge (Cloudflare)

- **Role:** The Shield.
- **Why it is critical:** our server cannot handle the load of blocking thousands of bots. Cloudflare filters traffic *before* it reaches our AWS/Ubuntu instance.
- **Implemented Features:**
    - **WAF (Web Application Firewall):** automatically detects and blocks the "Directory Traversal" signatures seen in our logs.
    - **Bot Fight Mode:** Challenges suspicious User Agents (like l9scan) with JavaScript challenges, effectively stopping the "spray and pray" scanners from wasting our server's CPU.
    - **Geo-Blocking:** If our users are only in Egypt, Cloudflare can block traffic from high-risk countries (Russia, China, etc.), reducing noise by ~90%.

Layer 2: Application Security (Spring Security 6)

- **Role:** The Gatekeeper.
- **Why it is critical:** It ensures that even if a request gets past Cloudflare, it cannot access sensitive data without a valid key.
- **Implemented Architecture:**
    - **Stateless JWT (JSON Web Tokens):** We explicitly disabled HTTP Sessions (SessionCreationPolicy.STATELESS). This is the most important configuration for our low-RAM server. Instead of storing user data in memory (which would crash our server), we verify a cryptographically signed token with every request.

- **Role-Based Access Control (RBAC):**
  - /api/doctor/** is strictly locked to users with ROLE_DOCTOR.
  - /api/patient/** is locked to ROLE_PATIENT.
  - Public endpoints (Login/Register) are explicitly whitelisted.

- **JWT Filter Chain:** A custom JwtAuthenticationFilter intercepts requests before they hit the database, validating signatures and expiration times.

Layer 3: Application Hardening (Anti-Leakage)

- **Role:** The Mask. Ensures that even if the application fails, it reveals no technical details to attackers.
- **Implemented Configs:**
  - **Disabled Stack Traces:** set `server.error.include-stacktrace=never` to prevent Java code structure from appearing in API errors.
  - **Removed Server Header:** set `server.server-header=` to hide the fact that the server is running "Apache-Coyote/1.1".
  - **Locked Actuator:** Restricted `management.endpoints.web.exposure` to only show `health`, blocking access to environment variables (`env`) and memory dumps (`heapdump`).
  - **Global Exception Handling:** Implemented a `@RestControllerAdvice` class to catch all Java exceptions and return a sanitized, generic JSON response (e.g., *"An internal error occurred"*) instead of a raw stack trace.

---

3. Server Stability Fixes

The server initially crashed due to insufficient RAM for running both Oracle Database 21c and Spring Boot. We applied three critical resource fixes.

- **Swap Space:** Created a **4GB Swap File** to act as emergency memory, preventing immediate crashes when RAM fills up.
- **Shared Memory:** Increased `/dev/shm` to 4GB to satisfy Oracle's `MEMORY_TARGET` requirement.
- **Database Connection Pool Tuning (HikariCP):**
  - **Issue:** The default pool size (10 connections) caused "Thread Starvation" errors because the server could not handle the CPU context switching and memory overhead.

- **Fix:**
  - Reduced `maximum-pool-size` to **5**.
  - Increased `connection-timeout` to **60000ms (60s)** to prevent the app from panicking when the server is slow due to Swap usage.
  - Added `validation-timeout` to prevent `ORA-17008: Closed Connection` errors.

---

4. Critical Considerations & Next Steps

We have a working "Shield" and "Lock," but our server's hardware limitations introduce unique risks.

Memory Exhaustion (DoS Risk)

- **The Risk:** our server relies on a 4GB Swap file (disk-based RAM) to function. Swap is very slow. A "Denial of Service" (DoS) attack doesn't need to be complex; simply sending 100 requests per second could cause our server to swap heavily, spiking CPU Load to 70+ (as seen previously) and freezing the app.

- **Mitigation:**
  - **Rate Limiting:** we *must* configure Cloudflare to limit requests per IP (e.g., max 50 requests per minute).
  - **Code-Level Limiting:** Consider adding the Bucket4j library to Spring Boot for application-level throttling.

Summary

Our environment is now functionally secure against common automated threats.

1. **Attackers** are hitting the firewall.
2. **Cloudflare** absorbs the volume.
3. **Spring Security** enforces the rules.
4. **Swap Space** keeps the server alive during spikes.

Recommendation: our immediate next step should be setting up **Uptime Monitoring** (like UptimeRobot) to alert you if the server enters a "Swap Death Spiral" again so you can reboot it immediately.