Testing Policy Document

Team : CacheME
Project : Secure File Sharing Platform

Version 2

Table of Contents

Introduction

This document outlines the Testing Policy for the Secure File Sharing Platform. The purpose of this policy is to establish a clear framework for testing procedures, ensuring that all aspects of the application are thoroughly evaluated for functionality, performance, security, and reliability. The Secure File Sharing Platform enables users to securely share and store their files in the cloud, providing robust protection against unauthorized access and data breaches. Given the complexity of the system, which involves frontend and backend components, cloud storage integration, encryption mechanisms, and user authentication, a robust testing strategy is essential to maintaining the application's quality and safeguarding user data.

The testing policy includes, but is not limited to:

- Unit testing
- Integration testing
- End-to-end testing
- Performance testing
- Usability testing
- Security testing

The following sections provide detailed procedures, tools, and responsibilities associated with each testing phase, ensuring that all components of the Secure File Sharing Platform are validated before deployment.

Project Overview

CacheMe - The Secure File Sharing Platform (SFSP) - SFSP enables users to securely share and store their files in the cloud. Employing advanced encryption technologies, secure access controls, and seamless integration features, SFSP delivers user-friendly file management and collaboration through an intuitive and responsive website.

Unit, Integration and E2EE testing Policies

This section details the testing strategy for our software development initiatives, emphasizing Unit Testing, Integration Testing, and End-to-End (E2E) Testing. The aim is to uphold the quality, dependability, and consistency of the codebase by enforcing strict testing protocols throughout the development process. We prioritize automated testing to boost efficiency and precision, integrating tests into our continuous integration (CI) pipeline, which is set up via GitHub Actions. This automated testing is integral to the pipeline, blocking code merges into the master branch if tests do not pass, thereby supporting elevated code quality standards.

To sustain a solid testing structure, we target a minimum test coverage of 80% across the project. This policy is applicable to all contributors and is intended to promote ongoing enhancement and trust in the application's functionality. Code coverage is monitored using Codecov, with corresponding badges displayed in our GitHub repository.

Unit Testing Policy

Objective

The goal of unit testing is to confirm the performance of individual software components, ensuring that each function, module, or service operates correctly. This involves testing logic, error management, and edge cases across the various languages employed in the project.

Automation

All unit tests must be automated and fully incorporated into the GitHub Actions pipeline. This setup enforces testing standards and prevents code from merging into integration branches if tests fail.

Coverage

We strive for a minimum code coverage of 80% for unit testing. Every function or module must be subjected to unit testing, with overall coverage exceeding 80% to guarantee the codebase's reliability and accuracy.

Frameworks

We leverage different frameworks suited to the languages in our technology stack:

- **React Next.js Frontend**: Cypress for unit testing.
- **Express.js API**: Jest for unit testing.
- **Golang File Service**: Testify for unit testing.
- **Python Key Service**: PyTest for unit testing.

Integration Testing Policy

Objective

The aim of integration testing is to verify the interactions between various components within and across services, ensuring they operate cohesively as intended. This includes testing the frontend's engagement with the API, the API's interaction with backend services, and database connections.

Automation

All integration tests must be automated and fully integrated into the GitHub Actions pipeline.

Frameworks

- **Frontend Integration**: Cypress is used to perform integration tests on the React Next.js frontend, focusing on its interactions with the Express.js API.
- **API Integration**: Jest is applied to conduct integration tests within the Express.js API, confirming interactions with the Golang File Service and Python Key Service.
- **Backend Integration**: PyTest and Testify are employed to validate database connectivity and ensure proper integration between the Golang File Service, Python Key Service, and the database.

End-to-End (E2E) Testing Policy

Objective

The purpose of end-to-end (E2E) testing is to ensure that the entire workflow of the Secure File Sharing Platform performs as expected from beginning to end. This encompasses verifying user interactions, file upload and download processes, API responses, and the overall system functionality.

Framework

We use Cypress for conducting end-to-end testing, offering strong capabilities for simulating user actions and validating application workflows across the frontend, API, and backend services.

Automation

The end-to-end tests must be automated and included in the GitHub Actions pipeline as the final validation step to confirm the application’s functionality.

Testing Environment

Cypress tests are run within the CI/CD pipeline on GitHub Actions. This configuration enables thorough testing of the application, ensuring seamless operation across all components—frontend, API, backend services, and database.

Testing Workflow

Overview

The testing workflow outlines the systematic approach we take to ensure the quality and reliability of our software throughout the development lifecycle. This process integrates various testing types—unit, integration, and end-to-end—and leverages automation to enhance efficiency and accuracy, while incorporating a collaborative review process to validate test quality.

Steps in the Testing Workflow

1. Development:

Developers create code for new features or bug fixes. Alongside this, they develop corresponding unit tests to validate individual components.

2. Unit Testing:

Automated unit tests are executed in the GitHub Actions pipeline when a branch is being merged into an integration branch. The tests verify that individual functions work as expected and achieve the desired code coverage of at least 80%. Another team member independently tests the functionality, and two additional team members review and approve the tests before proceeding, ensuring high-quality validation.

3. Integration Testing:

Once unit tests pass and are approved, integration tests are conducted to validate the interactions between components. These tests run in a designated environment within the GitHub workflow to simulate real-world scenarios. A team member performs the initial testing, followed by review and approval from two other team members to confirm the tests meet standards.

4. End-to-End Testing:

After integration tests are successfully completed and approved, end-to-end tests are performed in the same CI/CD pipeline on GitHub Actions. This stage verifies that the entire application workflow functions correctly from a user perspective, including component presence, user interactions, and request/response integrity. A team member conducts the testing, with two others reviewing and giving the go-ahead.

5.Regression Testing

Following changes to existing functionality, particularly after Demo 3, regression testing was implemented to ensure stability. The process begins by identifying the specific functionality affected by the changes. Tests are then executed to compare outputs against expected results. If the output matches expectations, the existing tests remain unchanged. If discrepancies are found, the tests are updated accordingly. This step is reviewed and approved by two team members to maintain test integrity.

Review and Feedback:

Any identified issues are documented, and feedback is shared among team members. This collaborative approach ensures thorough inspection of all functionality and necessary adjustments are made before moving forward.

Deployment:

Once all testing phases are complete, issues resolved, and approvals granted, the application is deployed to the production environment by merging the code into the master branch, triggering the deployment workflow.

Non-functional Requirements Testing

Quality assurance is an important aspect of the software development lifecycle, focusing on ensuring that applications not only work correctly, but also meet the highest standards of performance in its entirety.

We prioritised testing for the following quality requirements:

- Performance(load and stress testing)
- Security
- Usability

Performance Testing

1)Load Testing

Purpose:The purpose of the load testing was to measure how the application performs under expected user load, ensuring it remains responsive for typical file-sharing scenarios.

- **Test Scenarios:**
  - File Upload: We simulated multiple users uploading files or varying sizes (1MB - 10MB) to test the /upload endpoint.
  - File Download: We simulated users downloading files to test the /download endpoint.
  - Authentication: We tested login and session validation as user must authenticate before file operations.
  - Mixed Workload: Combine upload, download and authenticate to mimic real-world usage with some intentional mis-clicks.
- **JMeter Setup:**
  - We used thread groups for each scenario
    - Threads: We used 10 virtual users to keep the laptop and server manageable and slowly increased to 50.
    - Ramp-Up Period: We started with 60 seconds to gradually add users, this avoided sudden spikes, but then we decreased it to 10 seconds which was more realistic for the server.
    - Loop Count: We set it to 5-10 iterations per user to simulate sustained activity.
  - We used Http request samplers:
    - For Upload: We configured a POST request to https://SecureShare.co.za/api/files/upload with 3 sample files(10MB, 50MB and 100MB). These included authentication headers.

        - For download we did a similar thing using a GET request for https://SecureShare.co.za/api/files/download?fileId=23233 with other parameters to make the request valid.
    - Listeners:
        - To track the response time we used listener to add a summary report and an aggregate report.
        - We also used a “View Result Tree” for debugging.
- **What we looked for**:
    - We looked at Response times, our aim was to have <2 seconds for authentication, <5 seconds for small uploads/downloads and <15 seconds for large files.
    - We also looked for throughput, which is measuring requests per seconds
    - The error rate, since we were mimicking a realistic simulation for users we aimed for <5% errors, these being wrong uploads, incorrect passwords etc..
    - The last thing we looked at was E2EE overhead, we know that the file decryption on the client side causes delays but we aimed for them to be as small as possible.

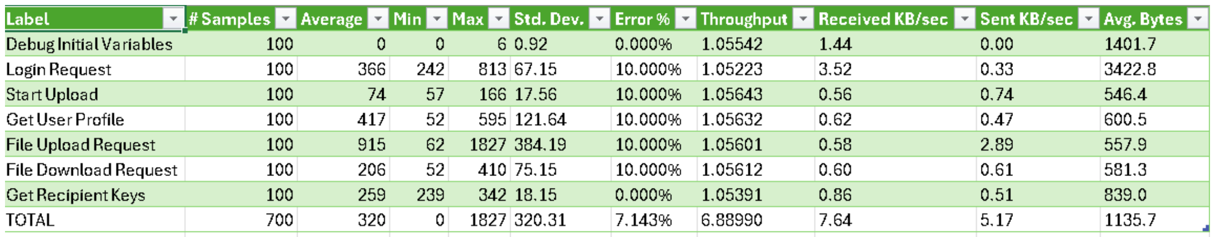
2) Stress Testing

The purpose: The purpose of stress testing was to identify the breaking point of the SecureShare application by subjecting it to extreme user loads and large file sizes, beyond typical usage scenarios. This helps determine the system’s capacity limits and scalability needs for handling peak file-sharing demands, ensuring reliability for critical group project operations.

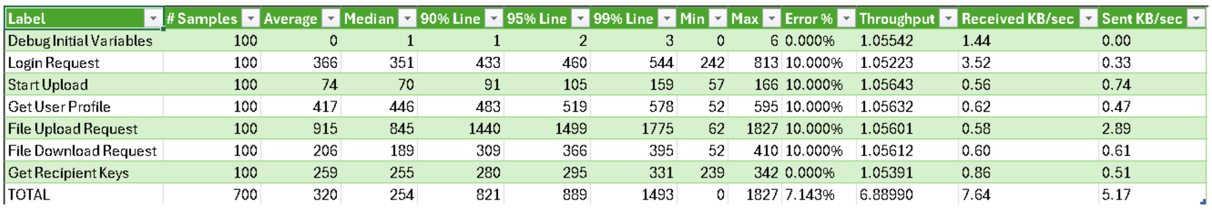
- **Testing Scenarios:**
    - High User Load: We simulated 50-100 concurrent users performing file uploads, downloads, and authentication to stress the server’s capacity.
    - Large File Sizes: Tested uploads and downloads of large files (100MB to 500MB) to evaluate storage, network, and E2EE processing limits.
    - Burst Traffic: We also simulated sudden spikes in activity (with 100 users uploading simultaneously) to mimic high-demand events, such as multiple team members sharing large files at once.
    - Mixed Workload with Errors: Combined uploads, downloads, and authentication with intentional errors (e.g., invalid file IDs, incorrect authentication tokens) to stress error-handling mechanisms.
- **JMeter Setup:**
    - Thread Group:
        - Threads: We started with 50 virtual users to keep tests manageable on the local laptop and hosted environment, incrementally increasing to 100. Tests were coordinated with the team to avoid disrupting the staging server at https://SecureShare.co.za.
        - Ramp-Up Period: Used a short ramp-up of 30 seconds for burst scenarios to simulate sudden traffic spikes, and 300 seconds for gradual stress tests to assess progressive load handling.
        - Loop Count: Set to 3 iterations per user to focus on peak load behavior, balancing test duration with laptop resource constraints.

- HTTP Request Samplers:
    - Upload: Configured a POST request to https://SecureShare.co.za/api/files/upload with sample files of 100MB and 500MB. Included authentication headers (e.g., Authorization: Bearer <token>) to mimic secure user sessions.
    - Download: Configured a GET request to https://SecureShare.co.za/api/files/download?fileId=23233 with valid parameters, testing 100MB and 500MB files.
    - Authentication: Configured a POST request to https://SecureShare.co.za/api/login with valid and invalid credentials to stress authentication services.
    - Error Simulation: Added requests with invalid file IDs (e.g., fileId=invalid123) and incorrect tokens to test error handling under load.
    - Constant Throughput Timer: Applied to maintain 100-200 requests per minute for high-load scenarios, ensuring consistent stress.
- Listeners:
    - Added “Aggregate Graph” to visualize response time spikes and error rates at breaking points.
    - Used “Summary Report” to track throughput and average response times.
    - Disabled “View Results Tree” during high-load tests to reduce laptop CPU/memory usage, enabling it only for initial debugging.
- **What we looked for:**

- **Breaking Point**: We identified when response times exceeded 15 seconds for uploads/downloads or 5 seconds for authentication, or when error rates surpassed 10%, indicating server overload.
- **Error Rate**: Targeted <10% errors, including 429 (Too Many Requests), 500 (Internal Server Error), or client-side E2EE decryption failures due to server delays.
- **Throughput**: Measured requests per second under stress, expecting a drop from load testing baselines (e.g., <5 req/s) as load increased.
- **Recovery Time**: Monitored how quickly the system returned to normal response times (<5 seconds) after reducing load.
- **E2EE Overhead**: Assessed whether client-side encryption/decryption (e.g., AES-GCM) caused significant delays under high load, aiming for minimal impact compared to load testing.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Debug Initial Variables	100	0	0	6	0.92	0.000%	1.05542	1.44	0.00	1401.7
Login Request	100	366	242	813	67.15	10.000%	1.05223	3.52	0.33	3422.8
Start Upload	100	74	57	166	17.56	10.000%	1.05643	0.56	0.74	546.4
Get User Profile	100	417	52	595	121.64	10.000%	1.05632	0.62	0.47	600.5
File Upload Request	100	915	62	1827	384.19	10.000%	1.05601	0.58	2.89	557.9
File Download Request	100	206	52	410	75.15	10.000%	1.05612	0.60	0.61	581.3
Get Recipient Keys	100	259	239	342	18.15	0.000%	1.05391	0.86	0.51	839.0
TOTAL	700	320	0	1827	320.31	7.143%	6.88990	7.64	5.17	1135.7



Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
Debug Initial Variables	100	0	1	1	2	3	0	6	0.000%	1.05542	1.44	0.00
Login Request	100	366	351	433	460	544	242	813	10.000%	1.05223	3.52	0.33
Start Upload	100	74	70	91	105	159	57	166	10.000%	1.05643	0.56	0.74
Get User Profile	100	417	446	483	519	578	52	595	10.000%	1.05632	0.62	0.47
File Upload Request	100	915	845	1440	1499	1775	62	1827	10.000%	1.05601	0.58	2.89
File Download Request	100	206	189	309	366	395	52	410	10.000%	1.05612	0.60	0.61
Get Recipient Keys	100	259	255	280	295	331	239	342	0.000%	1.05391	0.86	0.51
TOTAL	700	320	254	821	889	1493	0	1827	7.143%	6.88990	7.64	5.17

Security Testing

Purpose

The purpose of security testing was to identify vulnerabilities in the SecureShare file-sharing application, ensuring the end-to-end encrypted (E2EE) platform at `https://SecureShare.co.za` protects user data and maintains robust security. Using OWASP ZAP, we conducted passive, active, and manual tests to detect potential threats, with a focus on securing HTTP headers and reducing overall risk levels. The Zap application can be found at this site ZAP .

Test Methodology

- **Passive Scanning**: Analyzed traffic to `https://SecureShare.co.za` without sending intrusive requests, capturing issues like missing or misconfigured HTTP security headers, exposed metadata, and session management flaws.
- **Active Scanning**: Performed targeted probes on key endpoints (`/api/files/upload`, `/api/files/download`, `/api/login`) in a staging environment to identify vulnerabilities such as cross-site scripting (XSS), SQL injection, or broken access controls.
- **Manual Testing**: Conducted hands-on tests to validate E2EE implementation, focusing on client-side encryption (e.g., AES-GCM), key exchange (e.g., X3DH), and file access controls, ensuring no plaintext data was exposed.

Tests were run using OWASP ZAP in headless mode to minimize resource usage on the testing laptop, with the command: `zap.sh -cmd -quickurl https://SecureShare.co.za -quickout security_report.html`.

Key Findings

In the initial run, OWASP ZAP identified several HTTP headers that were either missing or misconfigured, posing potential security risks. While these issues were not immediate threats to the E2EE file-sharing functionality, they elevated the overall risk level to **High** due to potential exploitation in combination with other vulnerabilities.

- **Missing Headers**: Headers such as `Content-Security-Policy`, `X-Frame-Options`, and `Strict-Transport-Security` were absent, increasing risks of XSS, clickjacking, or HTTP downgrade attacks.

- **Misconfigured Headers**: Some headers, like `X-Content-Type-Options`, were present but incorrectly set, potentially allowing MIME-type sniffing.
- **Other Issues**: Passive scans flagged minor issues like verbose error messages and session cookie attributes lacking `Secure` or `HttpOnly` flags.

Remediation Efforts

We prioritized reducing the threat level from **High** to **Medium** or **Low** by addressing header-related vulnerabilities and other findings:

- **Header Fixes**:
  - Added `Content-Security-Policy: default-src 'self'` to restrict resource loading and mitigate XSS risks.
  - Implemented `X-Frame-Options: DENY` to prevent clickjacking.
  - Enabled `Strict-Transport-Security: max-age=31536000; includeSubDomains` to enforce HTTPS.
  - Corrected `X-Content-Type-Options: nosniff` to prevent MIME-type sniffing.
  - Updated session cookies to include `Secure` and `HttpOnly` attributes.
- **Active Scan Refinements**: Limited active scans to specific endpoints to reduce server load, confirming no critical vulnerabilities (e.g., SQL injection, XSS) after header fixes.
- **E2EE Validation**: Manual tests verified that file uploads/downloads via `/api/files/upload` and `/api/files/download` remained encrypted client-side, with no plaintext exposure on the server.

Post-remediation scans reduced the threat level to **Medium** for minor header misconfigurations and **Low** for informational findings, with no critical vulnerabilities remaining.

What We Looked For

- **Header Security**: Ensured all critical security headers were present and correctly configured to minimize risks.
- **E2EE Integrity**: Confirmed that client-side encryption (AES-GCM) and key exchange (X3DH) functioned correctly, with no data leaks during file transfers.
- **Endpoint Security**: Verified that `/api/files/upload`, `/api/files/download`, and `/api/login` resisted common attacks (e.g., XSS, unauthorized access).
- **Error Handling**: Checked that error messages did not expose sensitive information (e.g., stack traces, encryption keys).

Why

Security testing was critical to validate the SecureShare application's E2EE file-sharing functionality, ensuring robust protection against vulnerabilities. By addressing high-risk header issues and confirming E2EE integrity, we enhanced the platform's trustworthiness for secure group collaboration, aligning with the project's security-focused goals.

Make reference to our Zap Report.

Availability Testing

Objective

The objective was to ensure the website is consistently accessible, responsive, and performs reliably under various conditions, complementing Uptime Robot's monitoring of uptime and basic response.

Test Scope

- Focus: Website availability, server response time, and performance under load.
- Devices: Desktop, tablet, mobile.
- Browsers: Chrome, Firefox, Safari, Edge.
- Environments: Different network conditions (e.g., Wi-Fi, 4G, low-bandwidth).

Target Audience

- Tests are automated or conducted by the whole team; no end-user involvement is required.

Availability Test Plan

1. Uptime and Status Monitoring

Purpose: Verify the website is accessible 24/7 and detect any downtime beyond Uptime Robot's checks.

Tests:

1. Check availability of key pages such as dashboard.
2. Validate SSL certificate status and expiration.

Metrics:

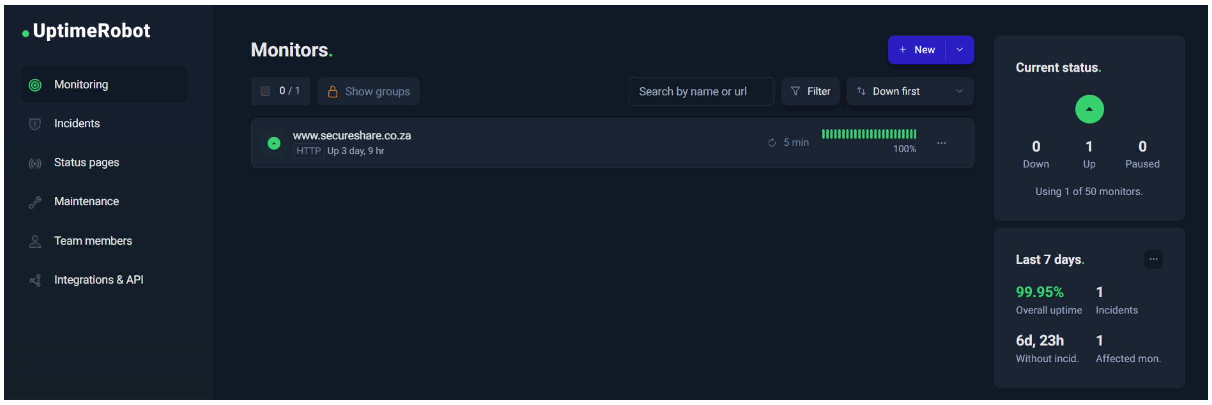
- Uptime percentage (target: ≥99.9%).
- Downtime incidents (frequency and duration).
- SSL certificate validity.

Procedure:

- Use Uptime Robot's existing checks (e.g., HTTP, ping, keyword monitoring).

- Supplement with tools like Pingdom or StatusCake for cross-region monitoring.
- Set alerts for downtime exceeding 5 minutes.
- Verify SSL status using tools like SSL Labs.

Results (19/10/2025):



Monitors.

www.secureshare.co.za
HTTP Up 3 day, 9 hr
5 min
100%

Current status.

0 Down
1 Up
0 Paused

Using 1 of 50 monitors.

Last 7 days.

99.95% Overall uptime
1 Incidents
6d, 23h Without incid.
1 Affected mon.