DevOps, Software Evolution and Software Maintenance

Group D

Group Members:

Victor Memborg-Heinrichsen	vmem@itu.dk
Peter Bjørholm Hansen	pbjh@itu.dk
Lukas Vranic	luvr@itu.dk
Anne-Marie Rommerdahl	annro@itu.dk

BSc Software Development
IT University of Copenhagen

BSDSESM1KU
May 9, 2025

Contents

1 Security Assessment

1.1 Risk Identification

Our assets:

- Codebase
- Database
- Logs

Threat Sources:

- SQL Injection
- Exposed ports (Prometheus)
- Vulnerable password hashing
- No requirements for user password (length, symbols, etc.)
- Vulnerable libraries
- No password on Dozzle (logging)

Risk Scenarios:

- A - Attacker performs SQL injection to download sensitive user data
- B - Attacker performs SQL injection to delete information from database
- C - Attacker performs SQL injection to log in on a legitimate user's account
- D - Attacker brute-forces the hash of a password, letting them log in on a legitimate user's account
- E - Attacker uses the open ports to overload our Prometheus with queries
- F - Attacker intersects messages sent over the network, as they are not encrypted
- G - Attacker exploits vulnerable library, which lets them perform some malicious attack
- H - Attacker accesses our logs on Dozzle, which lets them see all outputs, like usernames

1.2 Risk Analysis

Here are the scenarios presented on the risk matrix

	Rare	Unlikely	Likely	Certain
Catastrophic	B			G
Critical	A			G
Marginal			F	E,G
Negligible	C		D	G,H

Scenario A, B and C have a 'Rare' probability, as we already utilize the ORM-library GORM. GORM uses prepared statements and automatically escapes arguments to avoid SQL injection. Without GORM the probability would be much higher.
Scenario F was dealt with using the TLS protocol. Our Minitwit application using TLS is hosted at https://lukv.dk.

For scenario E, we would have to perform a workaround, as the exposed ports are caused by Docker and UFW being incompatible, an issue which has not yet been patched. However, Prometheus is the only service which is exposed, and the only threat is Prometheus crashing due to an overload of queries. Therefore, it is not considered a high-priority threat.
We could reduce the chance of scenario G happening, by keeping our libraries updated. We could also use tools that detect vulnerabilities in dependencies, and keep the dependencies updated automatically. Scenario H can be prevented by locking our Dozzle interface with a password. However, no sensitive data is logged in Dozzle, and thus it is considered low priority.

2 Scaling and Upgrades

2.1 Scaling

To scale our Minitwit application, we performed horizontal scaling using Docker Swarm. We created three manager nodes and four worker nodes.
We have replicas of the following services:

Service	Replicated/Global	Nr. of replicas
Minitwit App	Replicated	3
Minitwit API	Replicated	4
Prometheus	Global	N/A
Dozzle	Global	N/A

Prometheus and Dozzle are set to be global services, as they are responsible for monitoring and logging, respectively. In order not to miss critical information, we do monitoring and logging on all nodes.

2.2 Upgrades

To update the services in our swarm, we use the 'Blue-Green' update strategy for our Minitwit App and API. When there is an update, a new environment will be created with said update. When it is up and running, traffic will be redirected from the original environment to the newly created one. This leads to minimal downtime for our Minitwit.
In case something goes wrong, we do a rollback, to revert the changes made by the update. **why is rollback stop-first? does it matter?**

3 CI/CD Chain

We are using GitHub Actions to automate the testing and deployment of Minitwit. There are 5 GitHub Actions workflows:

1. Deploy services to DigitalOcean
2. Use Hadolint on dockerfiles
3. Run golangci-lint tool
4. Run static tools for staging
5. Release MiniTwit (automatically)

Use Hadolint on dockerfiles

This workflow only runs when it get called by other workflows, and analyse if the dockerfiles `docker/Dockerfile` and `docker/Dockerfile.api` contains any linting issues.
The steps for this workflow is:

1. **Checkout**
    (a) **Purpose**
        i. The workflow uses `actions/checkout@v2` to fetch the code from the repository.
    (b) **Steps**
        i. Checkouts the code.
2. **Hadolint Action app**
    (a) **Purpose**
        i. Checks for any linting errors using `hadolint/hadolint-action@v3.1.0` in `docker/Dockerfile`.
    (b) **Steps**
        i. Runs `hadolint-action` on `docker/Dockerfile`.
3. **Hadolint Action api**
    (a) **Purpose**
        i. Checks for any linting errors using `hadolint/hadolint-action@v3.1.0` in `docker/Dockerfile.api`.
    (b) **Steps**
        i. Runs `hadolint-action` on `docker/Dockerfile.api`.

Run golangci-lint tool

This workflow also only runs when it get called by other workflows, and analyses the go source code any linting issues.

1. **Environment setup**
    (a) **Purpose**
        i. Defines some environment variables that are used in the workflow.
    (b) **Steps**
        i. `GO_VERSION = stable`
        ii. `GOLANGCI_LINT_VERSION = v1.64`
2. **Detect Modules**
    (a) **Purpose**

        i. To output all the Go modules in `./minitwit`.

    (b) **Steps**

        i. Checkout the repository.

        ii. Runs `go list -m` to list all Go modules and outputs in JSON format.

3. **Format Go Files**

    (a) **Purpose**

        i. To format all the `.go` files, done by using `Jerome1337/gofmt-action@v1.0.5`.

    (b) **Steps**

        i. Checkout the repository.

        ii. Runs `gofmt-action` to verify formatting in `./minitwit`.

4. **Golangci Lint**

    (a) **Dependency**

        i. Needs *Detect Modules* to run.

    (b) **Steps**

        i. Checkouts the repository and setups a go environment.

        ii. Runs `golangci-lint` in each GO module.

Deploy services to DigitalOcean

This workflow runs every time the main branch gets a push.

1. **call-hadolint and call-golangci**

    (a) **Dependency**

        i. *Hadolint on dockerfiles* and *golangci-lint*.

    (b) **Purpose**

        i. to ensure code quality before deployment.

    (c) **Steps**

        i. Checkout the repository.

        ii. Run `Hadolint on dockerfiles`.

        iii. Run `golangci-lint`.

2. **Run tests**

(a) **Dependency**

i. `call-hadolint and call-golangci`.

(b) **Purpose**

i. To test the code to ensure code quality before deployment.

(c) **Steps**

i. Checkout the repository.

ii. Setup a Go environment with v1.22 of Go.

iii. Setup docker Compose.

iv. Find the test script (`run_tests.sh`).

v. Run the test script.

3. **Build And Deploy**

(a) **Dependency**

i. *Run tests*

(b) **Purpose**

i. Build a Docker image, push it to Docker Hub and Deploy it on Digital Ocean Droplet.

(c) **Steps**

i. Checkout the repository.

ii. Verify secrets.

iii. Login to Docker Hub.

iv. Set up Docker Buildx.

v. Build and push minitwit-app.

vi. Build and push minitwit-api.

vii. Configure SSH.

viii. Sync files with rsync.

ix. Deploy to server.

Release MiniTwit (automatically)

This workflow creates a release on Github every thursday at 23:30 UTC.

1. **Automatic Release**

(a) **Purpose**

i. Creates a release on Github.

(b) **Steps**

i. Checkout the repository.
ii. Fetch latest version tag.
iii. Determine next version.
iv. Validate against SemVar pattern.
v. Generate release notes.
vi. Create GitHub Release.

4 Monitoring

We monitored MiniTwit using Prometheus for collecting metrics and Grafana for visualizing them. This setup provided great insights into the application's performance and behavior.

4.1 Prometheus

Prometheus was integrated into both the API and the application through a custom middleware that intercepts HTTP requests to gather metrics. The metrics collected includes:

- `http_requests_total`: A counter that counts the total number of HTTP requests. It includes labels for request path, HTTP method, and status code. This metric helped us monitor request load per endpoint, track response statuses (in general or just on an endpoint basis).
- `http_request_duration_seconds`: A histogram measuring the time taken to process HTTP requests, in seconds. It is labeled by request path and method, enabling performance benchmarks for endpoints.
- `http_response_messages_total`: A counter that logs the total number of HTTP responses, categorized by status code and message type. Before logging was fully implemented, this metric was particularly helpful in identifying which endpoints triggered specific status messages and understanding the reasons behind them. A bit depricated as soon as logging was implemented.

4.2 Grafana

While Prometheus was important for collecting data, Grafana is utilized for visualizing the data. Grafana connects to Prometheus as a data source, enabling the creation of dashboards. As mentioned in section 4.1 we monitored both our API and app, which resulted in us setting two dashboard up; one for the app and one for the API.

For instance, the API dashboard converted the metrics collected by Prometheus into a clear visual representation of the API's health and performance. Panels were set up to showcase:

- Indicators from `http_requests_total`, such as the rate of requests per second, overall request load distributed by endpoint, and the ratio of successful and client-error status codes.
- Performance monitored by `http_request_duration_seconds`, including average response times and 99th percentile latency, allowing for quick identification of performance degradation.