OWASP Top 10 Report

Name: Tony Jiang

Semester: 6

Class: RB04

Contents

Introduction

This document aims to determine if the application addresses the OWASP Top 10 security risks. It will provide insights on how these risks will be covered in the application and assess whether this coverage is necessary in application.

Top 10 security risks

This outlines the overall security risks of the project. This documentation will include steps to mitigate each risk. Some risks are not yet scheduled and will be decided upon later.

	Likelihood	Impact	Risk	Action possible	Planned
A01:2021-Broken Access Control	High	High	High	Implement authentication and authorization in the authentication service, defining user roles and permissions to enforce proper access control, ensuring users can sign in and access only their authorized resources, such as personal details.	Yes, for sprint 5
A02:2021-Cryptographic Failures	Low	High	Low	Implement strong password hashing with salt in the user and authentication services to securely store passwords, ensuring that sensitive data is protected from cryptographic failures.	Done in sprint 4
A03:2021-Injection	Low	High	Low		Done for sprint 4

A04:2021-Insecure Design	High	High	High		N/A for sprint 5
A05:2021-Security Misconfiguration	Moderate	High	Moderate	Integrate Snyk into the development pipeline to detect security misconfigurations in Infrastructure as Code (IaC) files, identify vulnerabilities in dependencies, and ensure secure configurations.	Done for sprint 4 may add more for sprint 5
A06:2021-Vulnerable and Outdated Components	Low	High	Low	Integrate Snyk into the CI/CD pipeline for automated security testing, enabling continuous detection of outdated dependencies and vulnerabilities in the project.	Done for sprint 4
A07:2021-Identification and Authentication Failures	High	High	High		N/A for sprint 5
A08:2021-Software and Data Integrity Failures	High	High	High		N/A for sprint 5
A09:2021-Security Logging and Monitoring Failures	High	High	High		N/A for sprint 5
A10:2021-Server-Side Request Forgery	High	High	High		N/A for sprint 5

Sprint 4 Security Validation Report

Sprint goal: Enhance application security by addressing security misconfigurations, updating outdated components, and implementing secure password encryption.

Relevant top 10 items:

A02:2021-Cryptographic Failures

A03:2021-Injection

A05:2021-Security Misconfiguration

A06:2021-Vulnerable and Outdated Components

Cryptographic Failures

Objective

Implement bcrypt hashing with salt to securely store passwords in the user service, ensuring sensitive data is protected from cryptographic failures.

Explanation

Bcrypt is widely used and provides strong resistance against brute force attacks. Bcrypt first generates a salt, which is a random string of characters. This salt is then combined with the password. After that, bcrypt hashes the salted string, transforming it into a fixed-length string of characters. This is a one-way process, meaning once the data is hashed, it cannot be reversed to retrieve the original data.

For example, a bcrypt hash might look like this:
$2a$10$Xw30UXChR4t7pniUdxtC6OC.J7r2DRq1D0RGFXSTKOuxshikkHMZi

$2a$ identifies the bcrypt version. 10 is the cost factor, which determines how many iterations the algorithm performs during hashing. In this case, the cost factor of 10 means the algorithm will perform 2^10 (1024) iterations. The rest of the string is the salt and the resulting hash.

Acceptance criteria

- All passwords must be securely hashed using bcrypt with a unique salt for each password.
- The salt and hashed passwords must not be stored in plain text.
- Passwords must not be reversible; only the hashed version should be stored.

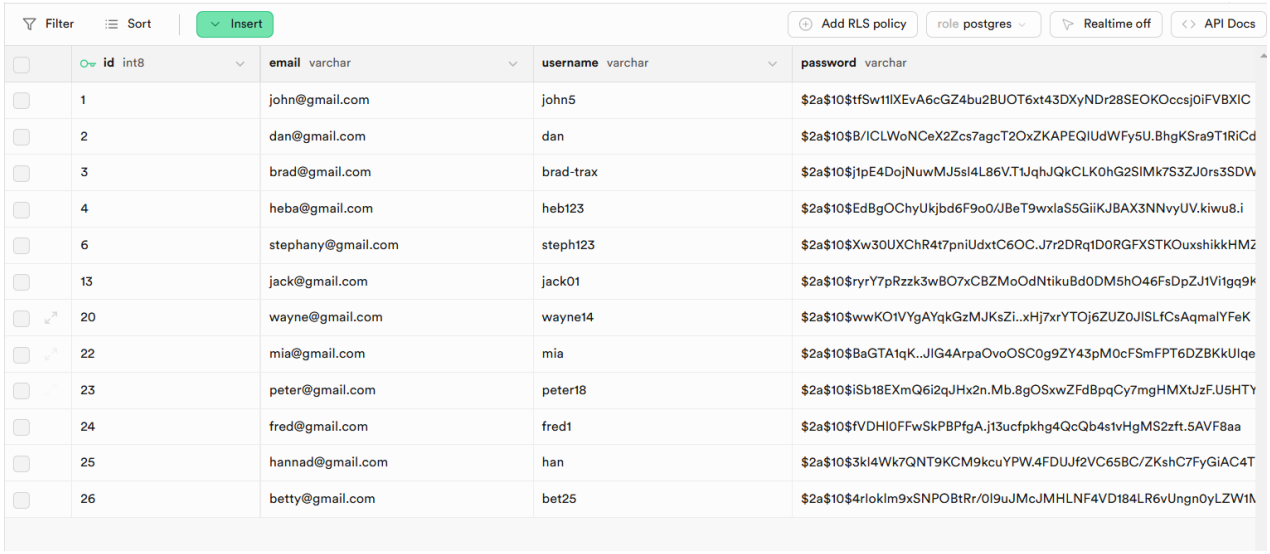
Validation method

- Code Review: Verify that the bcrypt hashing function is used in the registration process of the user service, ensure that the salt is generated uniquely for each password and used in conjunction with bcrypt for hashing. Additionally, the hashed password are store in the database.

Outcome

- Code Review: Confirmed that bcrypt is used for hashing passwords, with a unique salt for each password, and the hash is stored securely.

```go
import (
    "context"
    "errors"

    "regexp"

    "golang.org/x/crypto/bcrypt"

    "user-service/models"
    "user-service/repository"
)

func Register(ctx context.Context, user *models.User) error {
    // Hash the password using bcrypt
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)
    if err != nil {
        return errors.New("failed to hash password")
    }
```



Filter | Sort | Insert | Add RLS policy | role postgres | Realtime off | API Docs

id int8	email varchar	username varchar	password varchar
1	john@gmail.com	john5	$2a$10$tfSw11lXEvA6cGZ4bu2BUOT6xt43DXyNDr28SEOKOccsj0iFVBXIC
2	dan@gmail.com	dan	$2a$10$B/ICLWoNCeX2Zcs7agcT2OxZKAPEQlUdWFy5U.BhgKSra9T1RiCd
3	brad@gmail.com	brad-trax	$2a$10$j1pE4DojNuwMJ5sl4L86V.T1JqhJQkCLK0hG2SlMk7S3ZJ0rs3SDW
4	heba@gmail.com	heb123	$2a$10$EdBgOChyUkjbd6F9o0/JBeT9wxlaS5GiiKJBAX3NNvyUV.kiwu8.i
6	stephany@gmail.com	steph123	$2a$10$Xw30UXChR4t7pniUdxtC6OC.J7r2DRq1D0RGFXSTKOuxshikkHMZ
13	jack@gmail.com	jack01	$2a$10$ryrY7pRzzk3wBO7xCBZMoOdNtikuBd0DM5hO46FsDpZJ1Vi1gq9k
20	wayne@gmail.com	wayne14	$2a$10$wwKO1VYgAYqkGzMJKsZi..xHj7xrYTOj6ZUZ0JlSLfCsAqmalYFeK
22	mia@gmail.com	mia	$2a$10$BaGTA1qK..JlG4ArpaOvoOSC0g9ZY43pM0cFSmFPT6DZBKkUIqe
23	peter@gmail.com	peter18	$2a$10$iSb18EXmQ6i2qJHx2n.Mb.8gOSxwZFdBpqCy7mgHMXtJzF.U5HTY
24	fred@gmail.com	fred1	$2a$10$fVDHl0FFwSkPBPfgA.jt3ucfpkhg4QcQb4s1vHgMS2zft.5AVF8aa
25	hannad@gmail.com	han	$2a$10$3kl4Wk7QNT9KCM9kcuYPW.4FDUJf2VC65BC/ZKshC7FyGiAC4T
26	betty@gmail.com	bet25	$2a$10$4rloklm9xSNPOBtRr/0l9uJMcJMHLNF4VD184LR6vUngn0yLZW1N

Note

- Consider implementing automated testing to confirm that the password is hashed.
- Consider doing penetration testing to validate the strength of the bcrypt hash and its resistance to attacks for example brute-force or dictionary attack simulations test.

Injection

Objective

Prevent SQL injection by enforcing the use of prepared statements and parameterized queries for all database interactions.

Explanation

Parameterized query use placeholder in the SQL query and the value is supplied separately, for example in my case $1, $2, etc. This way, the database driver can distinguish between SQL code and the data values, preventing any potential SQL injection attacks. In my case the placeholder is $1, $2 because I use “github.com/jackc/pgx/v5” dependency in my Golang project.

```go
func CreateUser(ctx context.Context, user *models.User) error {
    query := `INSERT INTO users (email, username, password) VALUES ($1, $2, $3)`
    _, err := database.DB.Exec(ctx, query, user.Email, user.Username, user.Password)
    return err
}
```

Acceptance criteria

- Prepared statements or parameterized queries must be used for every database interaction.
- Inputs containing SQL symbols (', --, ;, etc.) must not alter the database query behavior.

Validation method

- Manually testing: Test input fields with SQL symbols and malicious payloads to confirm that queries remain unaffected.
- Code review: Verify that all database queries are parameterized and do not concatenate user input directly into SQL statements.

Outcome

- Manually testing: Still need to be tested.
- Code review: Confirmed all queries use prepared statements or parameterized queries to mitigate SQL injection risks.

```go
func CreateUser(ctx context.Context, user *models.User) error {
    query := `INSERT INTO users (email, username, password) VALUES ($1, $2, $3)`
    _, err := database.DB.Exec(ctx, query, user.Email, user.Username, user.Password)
    return err
}
```

Notes

- Consider adding automated SQL injection detection tools to the CI/CD pipeline for continuous validation.
- Implement centralized input validation and sanitization for all user inputs to ensure consistency and additional layers of protection.

Security Misconfiguration

Objective

Identify and remediate security misconfigurations in the application using the Snyk tool, ensuring secure default settings and configurations for the application, dependencies, and sensitive data handling.

Explanation

Snyk is a tool that detects outdated dependencies, dependency vulnerabilities, and misconfigured dependencies. It can also be integrated into the CI/CD pipeline for automated detection whenever you push or pull in your project.

Acceptance criteria

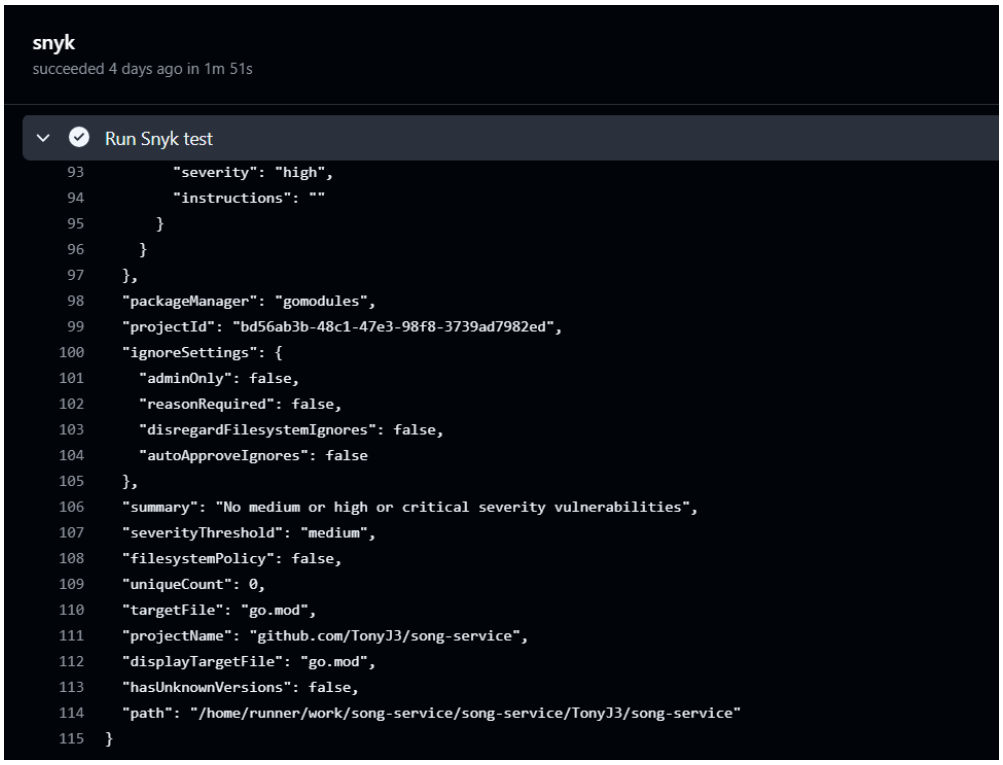
- All outdated or vulnerable dependencies and packages must be identified and updated.
- Sensitive environment variables and configuration files must be securely handled, ensuring they are not exposed or pushed to version control.

Validation method

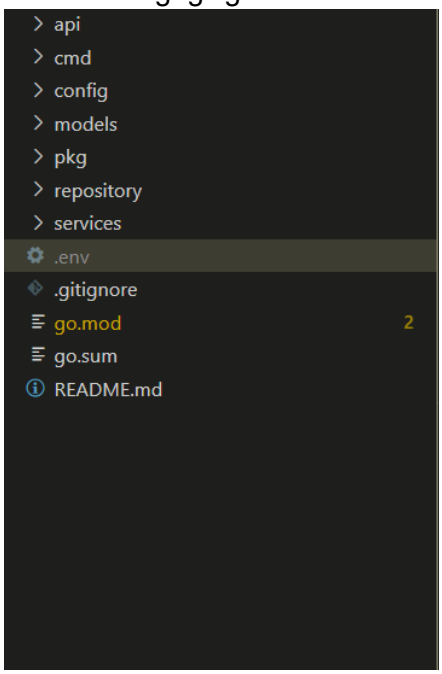
- Automated Snyk dependency scanning: Integrate Snyk into the CI/CD pipeline to automatically scan project dependencies for vulnerabilities caused by misconfigured or outdated components.
- Manual review: Review configuration files (e.g., .env, Dockerfile, config.yml) to ensure that hardcoded sensitive information such as secrets, API keys, or passwords are properly excluded from version control by using .gitignore rules.

Outcome

- Automated Snyk dependency scanning: Snyk was successfully integrated into the CI/CD pipeline. Vulnerability scans now run automatically on every push, identifying any misconfigured or vulnerable dependencies.

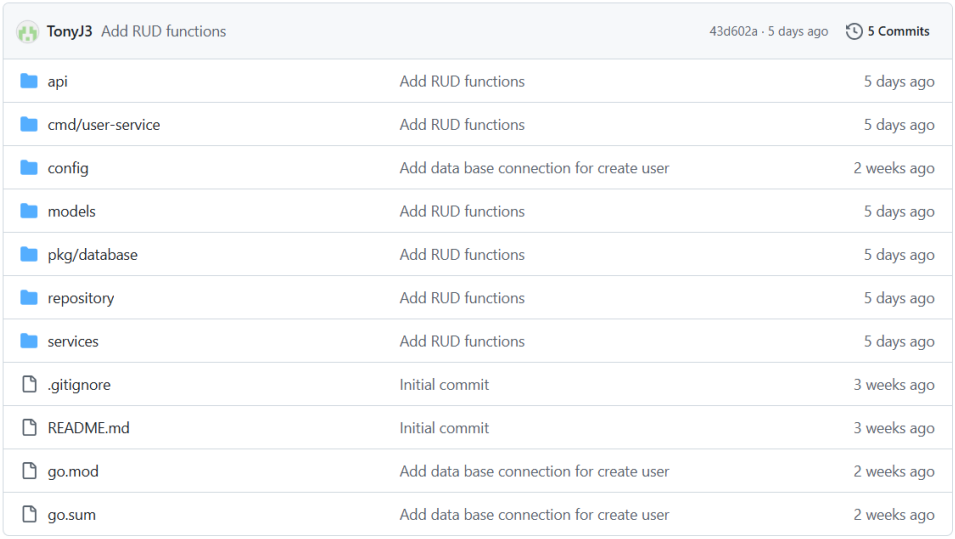


- Manual review: Confirmed that sensitive files like .env are properly excluded from version control using .gitignore.



.gitignore

```
*.exe~
*.dll
*.so
*.dylib

# Test binary, built with `go test -c`
*.test

# Output of the go coverage tool, specifically when used with LiteIDE
*.out

# Dependency directories (remove the comment below to include it)
# vendor/

# Go workspace file
go.work
go.work.sum

# env file
.env

```



Note

- Consider conducting a penetration test to identify additional misconfigurations, especially in runtime environments.
- Consider adding automated tools for scanning secrets
- Implement a central secret management solution to further secure sensitive data.

Vulnerable and Outdated Components

Objective

Implement the Snyk tool to detect outdated dependencies and vulnerabilities in project dependencies, ensuring that all components are up-to-date and secure.

Explanation

Snyk is a tool that detects outdated dependencies, dependency vulnerabilities, and misconfigured dependencies. It can also be integrated into the CI/CD pipeline for automated detection whenever you push or pull in your project.

Acceptance criteria

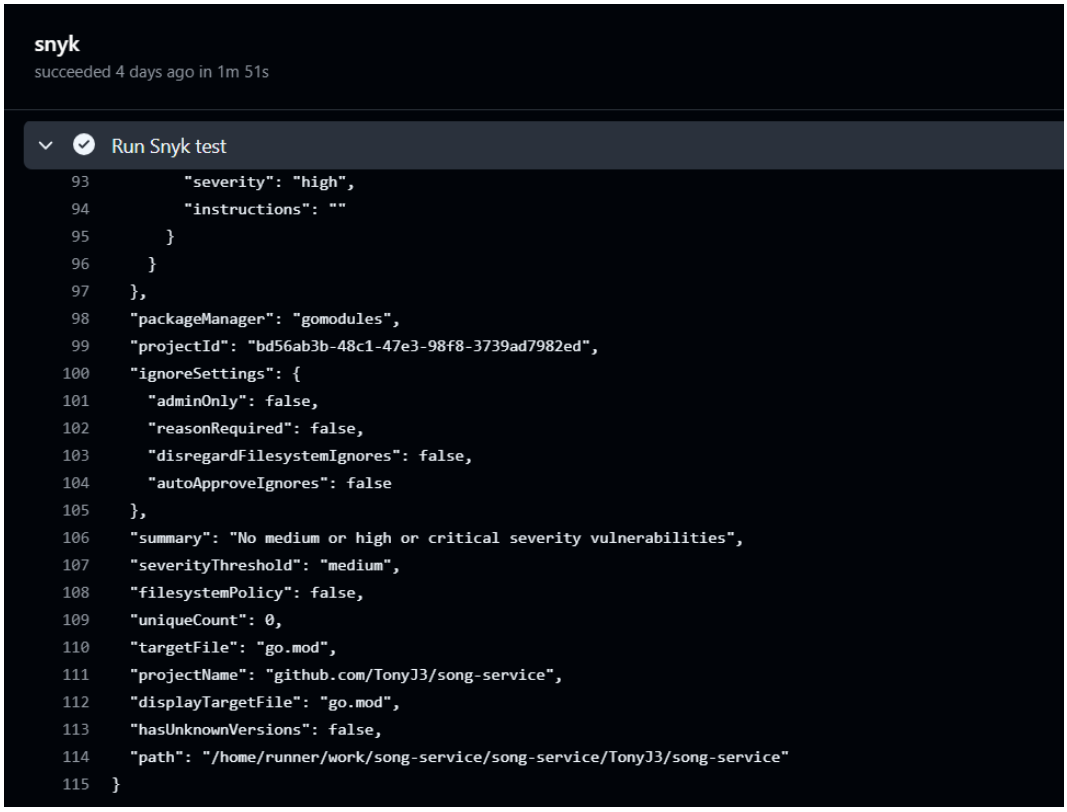
- All outdated or vulnerable dependencies and packages must be identified and updated to the latest secure versions.
- No critical vulnerabilities from outdated dependencies should remain in the project.

Validation method

- Automated Snyk dependency scanning: Integrate Snyk into the CI/CD pipeline to automatically scan project dependencies for vulnerabilities caused by outdated or insecure components.

Outcome

- Automated Snyk dependency scanning: Snyk was successfully integrated into the CI/CD pipeline, and vulnerability scans now run automatically with each push. Vulnerabilities due to outdated dependencies are flagged, and updates are required for identified issues.



Note

Consider implementing automated scanning for third party components for CI/CD.

Sprint 4 Security Validation Report