Øvelse 41 – Statisk sikkerhedsanalyse med CodeQL

Formål

Formålet med øvelsen er at opnå praktisk erfaring med at integrere statisk sikkerhedsanalyse i en Continuous Integration-pipeline.

I øvelse 38–40 har I konfigureret GitHub Actions til at bygge applikationen, udføre unit tests og kontrollere NuGet-afhængigheder for kendte sårbarheder.

Nu skal I udvide workflowet med **CodeQL**, som kan analysere applikationens egen kildekode for potentielle sikkerhedsproblemer.

Efter øvelsen skal I kunne:

- Forklare formålet med Static Application Security Testing (SAST).
- Konfigurere CodeQL i GitHub Actions.
- Forklare, hvordan CodeQL analyserer C#-kode.
- Undersøge og fortolke fund fra statisk kodeanalyse.
- Skelne mellem CodeQL-fund og fund fra NuGet Audit.
- Forklare begrænsninger ved automatiseret sikkerhedsanalyse.

Baggrund

I de foregående øvelser har vi etableret automatiske kontroller, som kan undersøge:

1. Om applikationen kan kompileres.
2. Om applikationens unit tests gennemføres.
3. Om tredjepartsafhængigheder indeholder kendte sårbarheder.

Disse kontroller er vigtige, men de undersøger ikke nødvendigvis sikkerheden i applikationens egen kildekode.

En applikation kan eksempelvis have alle tests gennemført og anvende opdaterede afhængigheder, men stadig indeholde SQL Injection eller Path Traversal.

Static Application Security Testing (SAST)

Static Application Security Testing (SAST) er statisk sikkerhedsanalyse af applikationens kildekode. Analysen gennemføres uden at skulle starte applikationen og sende HTTP-requests til den.

SAST-værktøjer kan anvende forskellige analyseteknikker til at identificere potentielle sikkerhedsproblemer.

Eksempelvis kan analysen undersøge:

- **Usikre kodemønstre:** Anvendelse af potentielt usikre metoder, manglende parameterisering af SQL-forespørgsler eller andre kodekonstruktioner, som kan introducere sikkerhedssårbarheder.
- **Dataflow og taint-analyse:** Hvordan potentielt usikkert input bevæger sig gennem applikationen og anvendes i sikkerhedskritiske operationer.

Et eksempel på dataflowanalyse er **taint-analyse**, som I tidligere har arbejdet med i undervisningen.

Ved taint-analyse undersøger man blandt andet, hvordan data bevæger sig fra:

- **Source:** En kilde til potentielt usikkert input, eksempelvis en HTTP-parameter.
- **Sink:** En sikkerhedskritisk operation, eksempelvis en databaseforespørgsel eller filoperation.

Hvis et potentielt usikkert input kan nå en sikkerhedskritisk operation uden tilstrækkelig beskyttelse, kan der være en sårbarhed.

CodeQL er et SAST-værktøj, som både kan identificere usikre kodemønstre og analysere dataflows gennem et program.

I denne øvelse skal vi eksempelvis undersøge, om CodeQL kan identificere SQL Injection og Path Traversal i Order Management API'et.

Bemærk, at anvendelsen af en potentielt usikker metode ikke nødvendigvis er en sårbarhed i sig selv. Det afhænger blandt andet af, hvordan metoden anvendes, og om input er tilstrækkeligt beskyttet.

Kendte sikkerhedsproblemer i projektet

Order Management API'et indeholder bevidst introducerede sikkerhedssårbarheder.

Vi vil især undersøge følgende:

Sårbarhed	CodeQL-regel
SQL Injection	`cs/sql-injection`

Sårbarhed	CodeQL-regel
Path Traversal	`cs/path-injection`
Information Disclosure	`cs/information-exposure-through-exception`

Vi skal nu undersøge, hvilke fund GitHub Actions rapporterer.

Forudsætninger

I skal have gennemført øvelse 38, 39 og 40.

Det betyder, at I har:

- En fork af undervisningsprojektet på GitHub.
- Projektet klonet lokalt.
- En fungerende GitHub Actions-pipeline.
- Automatiseret build og unit tests.
- Scanning af NuGet-afhængigheder.

I skal fortsætte med det samme repository og den eksisterende workflow-fil:

`.github/workflows/ci.yml`

Jeres fork skal være offentlig, så GitHubs code scanning-funktion er tilgængelig uden et særskilt GitHub Code Security-abonnement.

I skal ikke installere CodeQL lokalt. GitHub Actions henter de nødvendige værktøjer, når workflowet udføres.

Del 1 – Undersøg de sikkerhedsproblemer, CodeQL skal finde

Inden vi konfigurerer CodeQL, skal vi undersøge nogle af de kodekonstruktioner, som analysen skal identificere.

Åbn:

`OrderManagement.Api/Controllers/OrdersController.cs`

Trin 1 – Find SQL Injection

Find den metode, som implementerer søgning efter ordrer.

Undersøg følgende kode:

```
var sql = $"SELECT * FROM Orders WHERE CustomerName = '{customerName}'";

var results = await db.Orders
    .FromSqlRaw(sql)
    .ToListAsync();
```

Inputparameteren `customerName` anvendes direkte til at konstruere en SQL-forespørgsel.

Hvis brugeren kan påvirke SQL-forespørgslens syntaks, kan det medføre SQL Injection.

Trin 2 – Find Path Traversal

Find metoden, som henter dokumenter.

Undersøg følgende kode:

```
var path = Path.Combine(
    environment.ContentRootPath,
    "Documents",
    fileName);

var content = await System.IO.File.ReadAllTextAsync(path);
```

Inputparameteren `fileName` indgår direkte i konstruktionen af en filsti.

Hvis stien ikke kontrolleres korrekt, kan en bruger potentielt påvirke, hvilke filer applikationen forsøger at læse.

Refleksion

1. Hvad er kilden til det potentielt usikre input i de to eksempler?
2. Hvilke sikkerhedskritiske operationer anvender inputtet?
3. Hvordan adskiller disse problemer sig fra en sårbar tredjepartsafhængighed?

I skal ikke rette sårbarhederne i denne del.

Del 2 – Udvid GitHub Actions med CodeQL

Vi skal nu tilføje CodeQL til den eksisterende CI-pipeline.

CodeQL får sit eget job, så almindelig build, test og dependency scanning kan udføres uafhængigt af den statiske kodeanalyse.

Lokal CodeQL-analyse

CodeQL kan også anvendes lokalt gennem **CodeQL CLI**, ligesom I tidligere har arbejdet med statisk kodeanalyse ved hjælp af Snyk CLI.

Den lokale opsætning af CodeQL er dog lidt mere omfattende. Den indebærer blandt andet installation af CodeQL CLI, oprettelse af en analysedatabase og udførelse af sikkerhedsanalyser.

Et eksempel på, hvordan CodeQL kan installeres og anvendes lokalt, findes i filen `Testing_vulnerabilities.md` i undervisningsprojektet.

I denne øvelse fokuserer vi på at **automatisere CodeQL-analysen gennem GitHub Actions**, så kildekoden analyseres automatisk, når ændringer overføres til GitHub.

Trin 1 – Åbn workflowet

Åbn:

`.github/workflows/ci.yml`

Workflowet indeholder allerede jobbet `build` fra de tidligere øvelser.

Trin 2 – Tilføj et CodeQL-job

Tilføj følgende job under `jobs`, på samme indrykningsniveau som `build`.

```yaml
  codeql:
    name: CodeQL security analysis
    runs-on: ubuntu-latest

    permissions:
      contents: read
      security-events: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v7

      - name: Setup .NET
        uses: actions/setup-dotnet@v6
        with:
          dotnet-version: '10.0.x'

      - name: Initialize CodeQL
        uses: github/codeql-action/init@v4
        with:
          languages: csharp
          build-mode: manual
          queries: security-extended

      - name: Restore dependencies
```

```
        run: dotnet restore

      - name: Build for CodeQL
        run: dotnet build --no-restore --no-incremental

      - name: Perform CodeQL Analysis
        uses: github/codeql-action/analyze@v4
```

Bemærk: `codeql` skal være et selvstændigt job under `jobs`. Det må ikke placeres inde i `build`-jobbets `steps`.

Trin 3 – Kontroller det samlede workflow

Jeres `.github/workflows/ci.yml` skal nu se således ud:

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

permissions:
  contents: read

jobs:
  build:
    name: Build, test and dependency scanning
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v7

      - name: Setup .NET
        uses: actions/setup-dotnet@v6
        with:
          dotnet-version: '10.0.x'

      - name: Restore dependencies
        run: dotnet restore -p:NuGetAudit=true -p:NuGetAuditMode=all

      - name: Build solution
        run: dotnet build --no-restore

      - name: Run unit tests
        run: dotnet test --no-build

      - name: Scan NuGet dependencies
        run: dotnet package list --vulnerable --include-transitive

  codeql:
    name: CodeQL security analysis
    runs-on: ubuntu-latest
```

```yaml
    permissions:
      contents: read
      security-events: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v7

      - name: Setup .NET
        uses: actions/setup-dotnet@v6
        with:
          dotnet-version: '10.0.x'

      - name: Initialize CodeQL
        uses: github/codeql-action/init@v4
        with:
          languages: csharp
          build-mode: manual
          queries: security-extended

      - name: Restore dependencies
        run: dotnet restore

      - name: Build for CodeQL
        run: dotnet build --no-restore --no-incremental

      - name: Perform CodeQL Analysis
        uses: github/codeql-action/analyze@v4
```

Gem filen.

Forventet resultat

Workflowet indeholder nu to jobs:

Job	Formål
`build`	Restore, build, unit tests og NuGet Audit
`codeql`	Statisk sikkerhedsanalyse af C#-koden

Da vi ikke har angivet en afhængighed mellem de to jobs med `needs`, kan GitHub Actions udføre dem parallelt.

Hvert job kører på sin egen runner og skal derfor selv hente kildekoden og klargøre sit buildmiljø.

Del 3 – Analyser CodeQL-konfigurationen

Vi skal nu undersøge, hvad de enkelte dele af CodeQL-jobbet gør.

1. CodeQL-jobbet

```yaml
codeql:
  name: CodeQL security analysis
  runs-on: ubuntu-latest
```

Her definerer vi et selvstændigt job til statisk sikkerhedsanalyse.

Jobbet udføres på en Ubuntu-runner.

2. Rettigheder

```yaml
permissions:
  contents: read
  security-events: write
```

`contents: read`

Giver læseadgang til repositoryets indhold.

`security-events: write`

Giver workflowet mulighed for at uploade analyseresultater til GitHubs code scanning-funktion.

Rettigheden er nødvendig i vores CodeQL-konfiguration.

Bemærk, at den udvidede rettighed kun gives til `codeql`-jobbet. Det almindelige `build`-job beholder den mere begrænsede rettighed fra workflowets overordnede konfiguration.

Dette understøtter sikkerhedsprincippet **least privilege**.

3. Initialisering af CodeQL

```yaml
- name: Initialize CodeQL
  uses: github/codeql-action/init@v4
  with:
    languages: csharp
    build-mode: manual
    queries: security-extended
```

Dette step klargør CodeQL til at analysere kildekoden.

`languages: csharp`

Angiver, at vi ønsker at analysere C#.

`build-mode: manual`

Angiver, at vi selv bestemmer, hvordan projektet skal bygges under CodeQL-analysen.

CodeQL kan anvende buildprocessen til at indsamle oplysninger om C#-koden.

`queries: security-extended`

Angiver, at vi ønsker at anvende CodeQLs udvidede sikkerhedsregler.

Disse regler omfatter flere sikkerhedskontroller end det almindelige standardsæt.

4. Build for CodeQL

```
- name: Restore dependencies
  run: dotnet restore

- name: Build for CodeQL
  run: dotnet build --no-restore --no-incremental
```

Når vi anvender manuel buildtilstand, skal kildekoden bygges mellem initialisering og analyse.

`dotnet restore` gendanner projektets afhængigheder.

`dotnet build --no-restore --no-incremental` udfører en buildproces uden at genbruge tidligere kompileringsresultater.

Dette giver CodeQL mulighed for at observere buildprocessen og indsamle oplysninger om den kode, der bliver kompileret.

5. Udfør analysen

```
- name: Perform CodeQL Analysis
  uses: github/codeql-action/analyze@v4
```

Dette step afslutter CodeQL-analysen og uploader resultaterne til GitHub.

Fundene kan efterfølgende undersøges gennem repositoryets code scanning-funktion.

Refleksion

1. Hvorfor har CodeQL sit eget job?
2. Hvorfor skal CodeQL-jobbet hente kildekoden igen?
3. Hvorfor angiver vi `security-events: write`?
4. Hvad betyder `build-mode: manual`?

5. Hvorfor er build-steppet placeret mellem initialisering og analyse?
6. Hvad er forskellen mellem `security-extended` og det almindelige standardsæt af sikkerhedsregler?

Del 4 – Commit og push CodeQL-konfigurationen

Vi skal nu overføre ændringerne til GitHub.

Åbn en terminal i projektets rodmappe.

Kør:

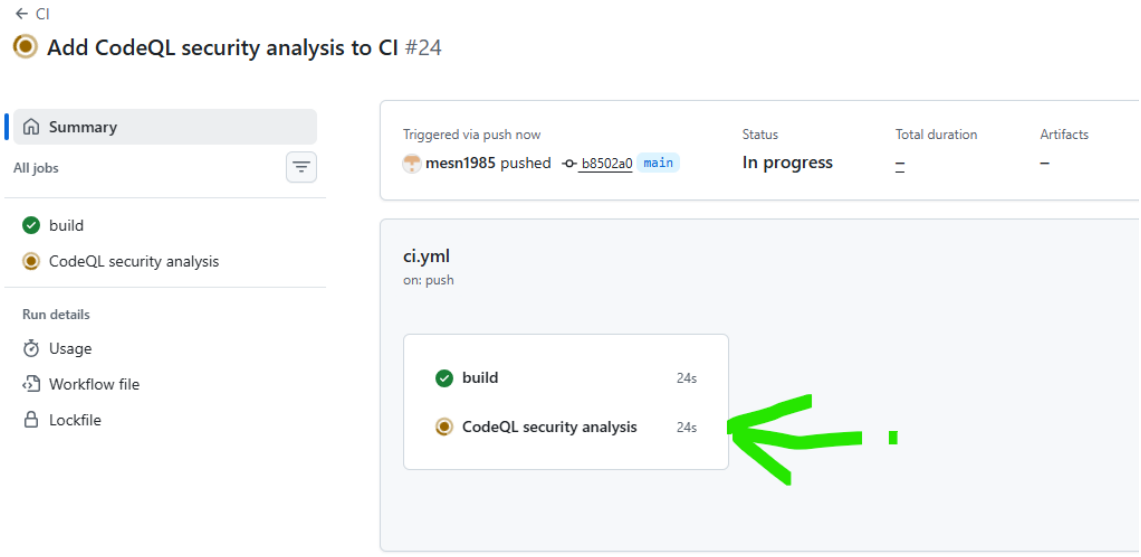
```
git add .github/workflows/ci.yml
git commit -m "Add CodeQL security analysis to CI"
git push origin main
```

Åbn derefter jeres repository på GitHub.

Vælg fanen **Actions**, og find det seneste workflow-run.

Forventet resultat

Workflowet indeholder nu både et `build`-job og et `codeql`-job.



CodeQL-jobbet skal blandt andet gennemføre følgende aktiviteter:

- Checkout repository.
- Setup .NET.

- Initialize CodeQL.
- Restore dependencies.
- Build for CodeQL.
- Perform CodeQL Analysis.

Hvis CodeQL ikke kan udføres

Kontroller først følgende:

- At repositoryet er offentligt.
- At GitHub Actions er aktiveret.
- At YAML-filen er korrekt indrykket.
- At `codeql` er placeret under `jobs`.
- At CodeQL-jobbet har rettigheden `security-events: write`.

Hvis GitHub har aktiveret CodeQL *default setup* på repositoryet, skal I undersøge konfigurationen under **Settings** → **Advanced Security / Code security and analysis**, så I ikke har to konkurrerende CodeQL-opsætninger.

Del 5 – Undersøg CodeQL-resultaterne

Vi skal nu undersøge, hvilke sikkerhedsproblemer CodeQL har identificeret.

Trin 1 – Undersøg workflow-loggen

Åbn det seneste workflow-run under **Actions**.

Klik på jobbet:

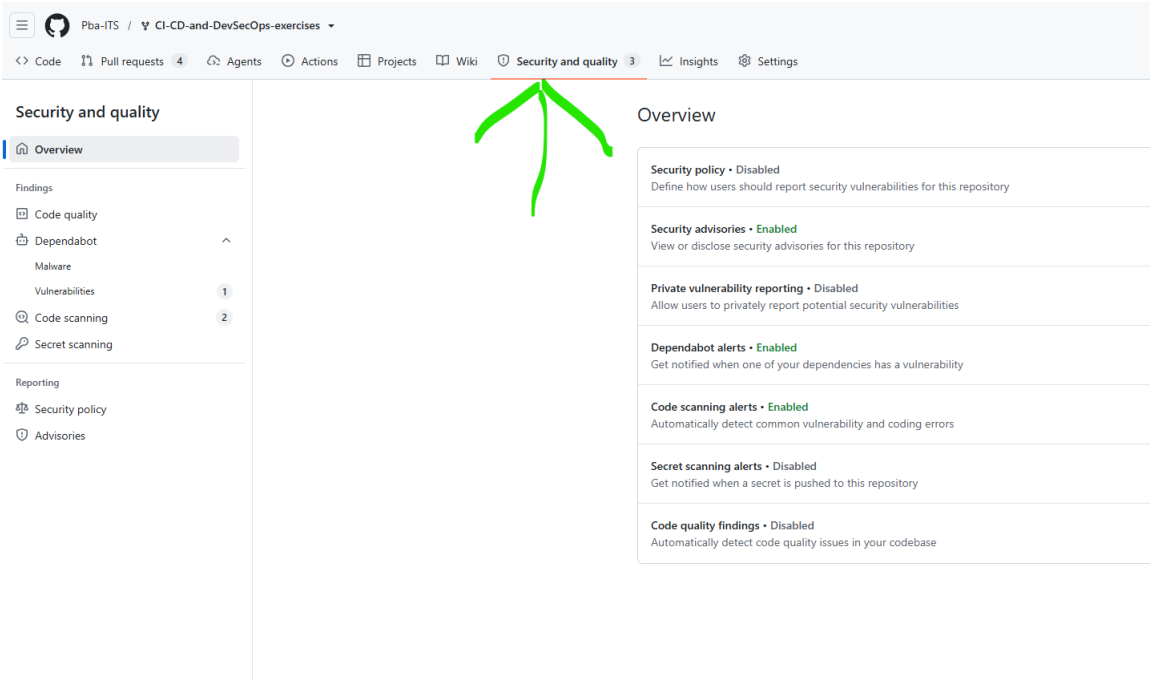
`CodeQL security analysis`

Kontroller, at initialisering, build og analyse er gennemført.

Trin 2 – Åbn code scanning-resultaterne

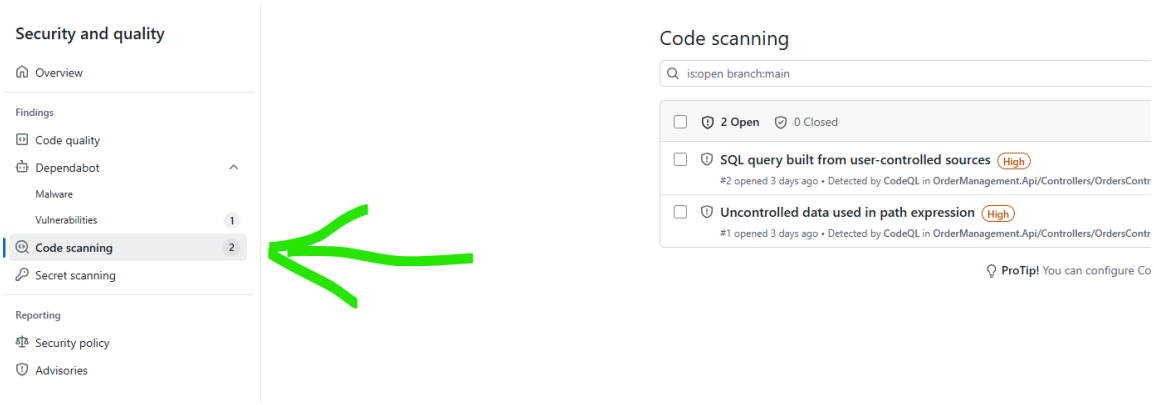
Åbn repositoryets fane:

Security and quality → Code scanning



Her kan I undersøge de sikkerhedsadvarsler, som CodeQL har rapporteret.

Gå ind på **Code scanning**



Trin 3 – Undersøg de identificerede fund

Find følgende CodeQL-regler:

Sårbarhed	CodeQL-regel	Tidligere lokalt resultat
SQL Injection	`cs/sql-injection`	Identificeret
Path Traversal	`cs/path-injection`	Identificeret

Sårbarhed	CodeQL-regel	Tidligere lokalt resultat
Information Disclosure	`cs/information-exposure-through-exception`	Ikke rapporteret

Ved lokal analyse af undervisningsprojektet med `security-extended` blev både SQL Injection og Path Traversal identificeret.

Vi forventer derfor, at CodeQL også kan rapportere disse fund i GitHub Actions.

Det præcise antal fund kan dog variere afhængigt af CodeQL-versionen og de anvendte regler.

Begrænsninger ved CodeQL

Det er vigtigt at forstå, at CodeQL ikke nødvendigvis identificerer alle sikkerhedssårbarheder i en applikation.

I vores undervisningsprojekt har vi eksempelvis en **Information Disclosure-sårbarhed**, som opstår på grund af usikker exception-håndtering:

```
catch (Exception ex)
{
    return StatusCode(500, ex.ToString());
}
```

Koden fanger en exception, men returnerer derefter interne oplysninger til klienten, herunder potentielt filstier, metodenavne og stack traces.

Sårbarheden er verificeret gennem en manuel HTTP-test, men **blev ikke identificeret af den lokale CodeQL-analyse med `security-extended`**.

CodeQL kan desuden have vanskeligt ved at identificere sikkerhedsproblemer, som skyldes fejl i applikationens **forretningslogik**.

Et eksempel kunne være en ordre, som fejlagtigt kan annulleres, selvom den allerede er leveret.

For at identificere denne type fejl kræves der kendskab til applikationens forretningsregler. En statisk kodeanalyse kan ikke nødvendigvis afgøre, om disse regler håndhæves korrekt.

Det betyder ikke, at CodeQL aldrig kan identificere fejl i forretningslogik, men at de almindelige sikkerhedsregler ikke nødvendigvis kender applikationens specifikke domæneregler.

Derfor skal CodeQL betragtes som et supplement til blandt andet:

- Manuel kodegennemgang.
- Unit tests og integrationstests.
- Dynamisk sikkerhedstest.
- Sikkerhedsvurdering af forretningsregler og misbrugstilfælde.

En succesfuld CodeQL-analyse er ikke en garanti for, at applikationen er sikker.

Trin 4 – Undersøg detaljer om fundene

Åbn fundene for SQL Injection og Path Traversal.

Undersøg følgende oplysninger:

- CodeQL-reglens ID.
- Filen, hvor problemet er identificeret.
- Den relevante kodelinje.
- Beskrivelsen af sikkerhedsproblemet.
- Eventuelle oplysninger om dataflow.
- Anbefalede sikkerhedsforanstaltninger.

Forventet resultat

De to sårbarheder forventes at være knyttet til filen:

`OrderManagement.Api/Controllers/OrdersController.cs`

Refleksion

1. Hvilke sikkerhedsfund blev rapporteret?
2. Hvor i kildekoden blev problemerne identificeret?
3. Hvordan beskriver CodeQL forbindelsen mellem brugerinput og den usikre operation?
4. Hvilken information giver CodeQL, som kan hjælpe en udvikler med at forstå problemet?

Del 6 – Sammenlign CodeQL og NuGet Audit

Vi har nu to forskellige sikkerhedsværktøjer i CI-pipelinen.

De undersøger forskellige dele af applikationens sikkerhed.

NuGet Audit

Undersøger projektets tredjepartsafhængigheder for kendte sikkerhedssårbarheder.

Eksempel:

`Newtonsoft.Json` version `12.0.1`.

CodeQL

Analyserer applikationens kildekode for potentielle sikkerhedsproblemer.

Eksempel:

```
var sql = $"SELECT * FROM Orders WHERE CustomerName = '{customerName}'";
```

Sammenligning

Egenskab	NuGet Audit	CodeQL
Primært analyseobjekt	Tredjepartsafhængigheder	Applikationens kildekode
Analysemetode	Sammenligning med kendte sårbarhedsoplysninger	Statisk programanalyse
Typiske fund	Sårbare pakkeversioner	Usikre kodekonstruktioner og dataflows
Eksempel fra projektet	`Newtonsoft.Json` 12.0.1	SQL Injection og Path Traversal

Refleksion

1. Hvorfor kan CodeQL ikke erstatte NuGet Audit?
2. Hvorfor kan NuGet Audit ikke erstatte CodeQL?
3. Hvilke sikkerhedsproblemer kan stadig være overset, selvom begge værktøjer har gennemført deres analyser?

Del 7 – Undersøg begrænsninger ved statisk kodeanalyse

En vigtig del af sikkerhedsarbejdet er at forstå, hvad de anvendte værktøjer kan og ikke kan identificere.

I Del 5 så vi, at Information Disclosure-sårbarheden ikke blev rapporteret ved den tidligere lokale CodeQL-analyse.

Vi skal nu sammenligne dette resultat med andre sikkerhedskontroller og undersøge, hvorfor forskellige analyseteknikker kan være nødvendige.

Trin 1 – Undersøg GitHubs resultater

Kontroller, om CodeQL har identificeret Information Disclosure i jeres repository.

Find den relevante kode i `OrdersController.cs`:

```
catch (Exception ex)
{
    return StatusCode(500, ex.ToString());
}
```

Notér, om CodeQL rapporterer et sikkerhedsproblem knyttet til denne kode.

Trin 2 – Overvej fejl i forretningslogik

I tidligere øvelser har I arbejdet med invarianter og unit testing.

Order Management API'et indeholder blandt andet en forretningsregel om, at en leveret ordre ikke efterfølgende må annulleres.

Forestil jer, at en udvikler ændrer forretningslogikken, så en leveret ordre alligevel kan annulleres.

Koden kan stadig være syntaktisk korrekt og kunne kompileres.

Den vil heller ikke nødvendigvis blive identificeret som usikker af CodeQL, fordi værktøjets almindelige regler ikke kender denne specifikke forretningsregel.

Overvej, hvordan en unit test kan anvendes til at opdage denne type fejl.

I skal ikke ændre applikationens forretningslogik i denne del.

Refleksion

1. Blev Information Disclosure identificeret af CodeQL?
2. Hvis ikke, betyder det så, at applikationen er fri for denne sårbarhed?
3. Hvorfor er kendskab til domæneregler vigtigt, når sikkerheden i forretningslogikken skal vurderes?

4. Hvordan kan unit tests understøtte verifikation af forretningsregler og invarianter?
5. Hvorfor er det relevant at kombinere statisk analyse, dynamisk sikkerhedstest og manuel kodegennemgang?
6. Betyder en grøn CI-pipeline, at applikationen er sikker?

Samlet refleksion

Besvar følgende spørgsmål med udgangspunkt i jeres egne observationer:

1. Hvordan adskiller statisk sikkerhedsanalyse sig fra unit testing?
2. Hvordan kan CodeQL identificere usikre dataflows i en applikation?
3. Hvilke sikkerhedsproblemer blev identificeret i Order Management API'et?
4. Hvilke begrænsninger har automatiseret statisk kodeanalyse?
5. Hvilken forskel er der på en sårbarhed i applikationens egen kode og i en tredjepartsafhængighed?
6. Hvordan understøtter kombinationen af automatiserede tests, NuGet Audit og CodeQL en mere sikker softwareudviklingsproces?

Resultat og evaluering

Øvelsen er gennemført, når I kan demonstrere følgende:

- I har udvidet den eksisterende CI-pipeline med CodeQL.
- CodeQL-jobbet gennemføres i GitHub Actions.
- I kan forklare de enkelte steps i CodeQL-jobbet.
- I har undersøgt CodeQLs sikkerhedsfund.
- I har kontrolleret, om SQL Injection og Path Traversal bliver identificeret.
- I har undersøgt, om Information Disclosure bliver rapporteret.
- I kan forklare forskellen mellem CodeQL og NuGet Audit.
- I kan forklare, hvorfor et succesfuldt CodeQL-job ikke betyder, at applikationen er sikker.

Det centrale er, at I kan **konfigurere automatiseret sikkerhedsanalyse, fortolke resultaterne og forstå værktøjets begrænsninger**.

I skal ikke rette de bevidst introducerede sårbarheder i denne øvelse.

Videre arbejde

Vi har nu etableret en CI-pipeline med:

1. Automatisk build.
2. Automatiserede unit tests.
3. Scanning af sårbare NuGet-afhængigheder.
4. Statisk sikkerhedsanalyse med CodeQL.

Disse kontroller hjælper med at identificere problemer under softwareudviklingen.

Men kendte sårbarheder i tredjepartsafhængigheder kan også blive offentliggjort, efter at en bestemt version af applikationen er udviklet.

I **øvelse 41a** skal vi derfor arbejde med **Dependabot**, som kan overvåge og foreslå opdateringer til projektets afhængigheder.

Litteratur og dokumentation

- *Secure by Design*, kapitel 8 – Leveraging your delivery pipeline for security.
- GitHub – CodeQL
- GitHub – CodeQL Action
- GitHub – CodeQL for compiled languages
- GitHub – Code scanning
- CodeQL – C# Query Help

Last update: 2026-11-10 20:57:39