Øvelse 40 – Scanning af sårbare afhængigheder med NuGet Audit

Formål

Formålet med øvelsen er at opnå praktisk erfaring med automatiseret scanning af tredjepartsafhængigheder som en del af en Continuous Integration-pipeline.

I øvelse 38 og 39 konfigurerede I GitHub Actions til automatisk at bygge Order Management API'et og udføre unit tests.

Nu skal I udvide den eksisterende CI-pipeline med en sikkerhedskontrol, der undersøger, om projektet anvender NuGet-pakker med kendte sikkerhedssårbarheder.

I skal bruge **NuGet Audit**, som er integreret i .NET SDK.

Efter øvelsen skal I kunne:

- Forklare, hvorfor 3.-partssårbarheder udgør en potentiel sikkerhedsrisiko.
- Anvende NuGet Audit til at identificere kendte sårbarheder.
- Skelne mellem direkte og transitive afhængigheder.
- Integrere scanning af afhængigheder i GitHub Actions.
- Fortolke sikkerhedsadvarsler og forstå deres betydning for CI-pipelinen.
- Forklare forskellen mellem en sikkerhedsadvarsel og en fejl, der stopper CI-processen.

Baggrund

Moderne softwareapplikationer anvender ofte biblioteker fra tredjeparter.

I .NET-projekter distribueres disse biblioteker typisk som NuGet-pakker.

Selvom applikationens egen kildekode fungerer korrekt, kan den indeholde sikkerhedsproblemer gennem de biblioteker, som den anvender.

Disse sikkerhedsproblemer betegnes **3.-partssårbarheder**, når sårbarhederne findes i softwarekomponenter udviklet af tredjeparter.

Håndtering af 3.-partssårbarheder er en del af det bredere område **software supply chain security**, som omhandler sikkerheden i softwareforsyningskæden, herunder afhængigheder,

udviklingsværktøjer og leveranceprocesser.

En afhængighed kan eksempelvis indeholde en sårbarhed, som først bliver opdaget og registreret i en offentlig sårbarhedsdatabase, efter at biblioteket er taget i brug.

NuGet Audit

NuGet Audit er en funktion i .NET og NuGet, som kontrollerer projektets afhængigheder mod oplysninger om kendte sikkerhedssårbarheder.

I .NET 10 udføres denne kontrol som udgangspunkt i forbindelse med `dotnet restore`.

Det betyder, at vores eksisterende CI-pipeline allerede kan vise sikkerhedsadvarsler under gendannelse af afhængigheder.

I denne øvelse gør vi sikkerhedskontrollen mere synlig ved at tilføje et særskilt step, som viser de sårbare pakker.

Direkte og transitive afhængigheder

Der skelnes mellem to typer afhængigheder:

Type	Beskrivelse
Direkte afhængighed	En pakke, som projektet selv refererer til
Transitiv afhængighed	En pakke, som indirekte indgår gennem en anden afhængighed

Begge typer kan introducere sikkerhedssårbarheder i applikationen.

Forudsætninger

I skal have gennemført øvelse 38 og 39.

Det betyder, at I allerede har:

- En fork af undervisningsprojektet.
- Et lokalt Git-repository.
- En fungerende CI-pipeline med GitHub Actions.
- Automatiseret restore, build og unit tests.

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

`.github/workflows/ci.yml`

I skal ikke oprette et nyt repository eller et nyt workflow.

Del 1 – Undersøg projektets NuGet-afhængigheder

Vi starter med at undersøge, hvilke tredjepartsbiblioteker applikationen anvender.

Trin 1 – Åbn projektfilen

Åbn:

`OrderManagement.Api/OrderManagement.Api.csproj`

Find projektets `PackageReference`-elementer.

En af de afhængigheder, der indgår i undervisningsprojektet, er:

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
```

Denne version er bevidst inkluderet, fordi den har kendte sikkerhedssårbarheder.

Trin 2 – Undersøg afhængighederne

Åbn en terminal i projektets rodmappe, hvor `OrderManagement.slnx` ligger.

Kør:

```
dotnet package list --include-transitive
```

Kommandoen viser projektets direkte og transitive NuGet-afhængigheder.

Forklaring

`dotnet package list`

Viser de NuGet-pakker, som projekterne anvender.

`--include-transitive`

Medtager også de pakker, som projektet afhænger af indirekte gennem andre pakker.

Refleksion

1. Hvad er forskellen mellem direkte og transitive afhængigheder?

2. Hvorfor kan en transitiv afhængighed udgøre en sikkerhedsrisiko?
3. Hvem har ansvaret for at håndtere kendte sårbarheder i tredjepartsbiblioteker?

Del 2 – Udfør NuGet Audit lokalt

Vi skal nu undersøge, om projektet indeholder afhængigheder med kendte sårbarheder.

Påmindelse om CVE'er og kendte sårbarheder

Som I tidligere har arbejdet med, kan **3.-partssårbarheder** være registreret med en **CVE-identifikator (Common Vulnerabilities and Exposures)**.

En CVE-identifikator bruges til entydigt at identificere en offentliggjort sikkerhedssårbarhed. Sårbarhedsbeskrivelsen angiver blandt andet, hvilke versioner af et softwareprodukt der er berørt.

NuGet Audit sammenligner projektets NuGet-pakker og deres versioner med oplysninger om kendte sårbarheder fra sikkerhedsdatabaser.

Sårbarhederne kan eksempelvis være dokumenteret gennem CVE-identifikatorer eller GitHub Security Advisories (GHSA).

NuGet Audit analyserer altså ikke bibliotekernes kildekode for nye sårbarheder, men undersøger, om de anvendte pakkeversioner har **allerede kendte og registrerede sårbarheder**.

Trin 1 – Udfør restore med NuGet Audit

Kør:

```
dotnet restore -p:NuGetAudit=true -p:NuGetAuditMode=all
```

Denne kommando gendanner projektets afhængigheder og udfører en sikkerhedskontrol.

Parametrene angiver eksplicit, at NuGet Audit skal være aktiveret, og at både direkte og transitive afhængigheder skal undersøges.

I .NET 10 er dette normalt allerede standardadfærden for et projekt, der anvender `net10.0`.

Trin 2 – Undersøg advarslerne

Undersøg output fra kommandoen.

Find eventuelle advarsler om kendte sårbarheder.

NuGet anvender følgende advarselskoder:

Advarselskode	Alvorlighedsgrad
`NU1901`	Low
`NU1902`	Moderate
`NU1903`	High
`NU1904`	Critical

Disse advarsler angiver, at en pakke har en kendt sårbarhed med den pågældende alvorlighedsgrad.

Trin 3 – Vis sårbare afhængigheder

Kør:

```
dotnet package list --vulnerable --include-transitive
```

Denne kommando viser de afhængigheder, hvor der er registreret kendte sårbarheder.

Forventet resultat

I skal kunne identificere den bevidst sårbare afhængighed:

`Newtonsoft.Json` version `12.0.1`.

Den er blandt andet berørt af en kendt sårbarhed knyttet til håndtering af stærkt indlejrede JSON-strukturer, som kan medføre Denial of Service (DoS).

Den relevante sikkerhedsadvarsel er knyttet til:

`CVE-2024-21907`

De konkrete advarsler og sårbarhedsreferencer afhænger af de oplysninger, der er tilgængelige fra NuGets auditkilder på tidspunktet for scanningen.

Refleksion

1. Hvilken pakke og version blev identificeret?
2. Hvilken alvorlighedsgrad har sårbarheden?

3. Hvilken information indeholder linket til sårbarhedsbeskrivelsen?
4. Betyder det nødvendigvis, at applikationen kan udnyttes gennem denne sårbarhed, fordi pakken er installeret?

Del 3 – Udvid GitHub Actions med dependency scanning

Nu skal vi gøre scanning af afhængigheder til en tydelig del af CI-pipelinen.

Åbn:

`.github/workflows/ci.yml`

Vi skal foretage to ændringer:

1. Opdatere det eksisterende restore-step, så NuGet Audit konfigureres eksplicit.
2. Tilføje et nyt step, som viser sårbare afhængigheder.

Trin 1 – Opdater Restore dependencies

Find det eksisterende step:

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

Ændr det til:

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

Vi aktiverer dermed eksplicit NuGet Audit og angiver, at både direkte og transitive afhængigheder skal undersøges for kendte sårbarheder.

I .NET 10 er dette normalt allerede standardadfærden, men den eksplicitte konfiguration gør sikkerhedskontrollen synlig i CI-pipelinen.

Trin 2 – Tilføj et nyt step

Tilføj følgende step efter `Run unit tests`:

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

Dette step viser de afhængigheder, hvor der er registreret kendte sikkerhedssårbarheder.

Trin 3 – Kontroller det samlede workflow

Jeres workflow skal nu se således ud:

```yaml
name: CI

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

permissions:
  contents: read

jobs:
  build:
    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
```

Gem filen.

Forklaring af de to ændringer

Restore dependencies

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

Dette step gendanner NuGet-afhængighederne og udfører NuGet Audit med den valgte konfiguration.

Sårbare afhængigheder kan blive rapporteret gennem NuGets advarselskoder.

Scan NuGet dependencies

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

Dette step giver en oversigt over de afhængigheder, hvor der er registreret kendte sårbarheder.

Det gør det lettere at finde og undersøge sikkerhedsfundene i GitHub Actions-loggen.

CI-pipelinen indeholder nu

Step	Formål
Checkout repository	Henter kildekoden
Setup .NET	Klargør .NET 10 SDK
Restore dependencies	Gendanner pakker og udfører NuGet Audit
Build solution	Kompilerer løsningen
Run unit tests	Udfører automatiserede tests
Scan NuGet dependencies	Viser afhængigheder med kendte sårbarheder

Vigtigt: Det nye scan-step er primært en rapporteringskontrol. Det er ikke konfigureret til automatisk at stoppe CI-pipelinen, blot fordi en sårbar afhængighed bliver fundet.

Refleksion

1. Hvorfor kan det være en fordel at have et særskilt step til scanning af afhængigheder?
2. Hvad er forskellen mellem en sårbarhedsadvarsel og en kompileringsfejl?
3. Hvad betyder det for CI-processen, hvis en scanning finder en sårbarhed, men jobbet stadig får status `Success`?

Del 4 – Commit og push ændringerne

Vi skal nu overføre det opdaterede workflow til GitHub.

Åbn en terminal i projektets rodmappe.

Kør:

```
git add .github/workflows/ci.yml
git commit -m "Add NuGet vulnerability scanning to CI"
git push origin main
```

Åbn derefter jeres repository på GitHub.

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

Forventet resultat

Workflowet skal indeholde det nye step:

`Scan NuGet dependencies`

Når steppet udføres, skal det undersøge projektets NuGet-afhængigheder.

Del 5 – Analyser resultaterne fra GitHub Actions

Åbn det nyeste workflow-run i GitHub Actions.

Klik på jobbet `build`.

Trin 1 – Undersøg restore-steppet

Åbn:

`Restore dependencies`

Find eventuelle advarsler med koderne `NU1901`, `NU1902`, `NU1903` eller `NU1904`.

Trin 2 – Undersøg scanning-steppet

Åbn:

`Scan NuGet dependencies`

Undersøg resultatet.

I skal kunne finde oplysninger om sårbare afhængigheder.

Trin 3 – Sammenlign resultaterne

Sammenlign med resultatet fra den lokale scanning.

Undersøg:

- Om samme pakke identificeres.
- Om samme version rapporteres.
- Om alvorlighedsgraden er den samme.
- Om scanningen gennemføres.
- Om CI-jobbet får status `Success` eller `Failure`.

Forventet resultat

Den sårbare `Newtonsoft.Json`-afhængighed skal fremgå af scanningen, når de relevante sårbarhedsoplysninger er tilgængelige fra auditkilden.

Workflowet kan godt gennemføres med status `Success`, selvom det indeholder sikkerhedsadvarsler.

Det skyldes, at NuGet Audit som udgangspunkt rapporterer kendte sårbarheder som advarsler.

Refleksion

1. Hvad fortæller det os, hvis CI-pipelinen er grøn, men samtidig rapporterer en kendt sårbarhed?
2. Hvorfor er det vigtigt at undersøge indholdet af en sikkerhedsscanning og ikke blot dens status?
3. Hvilke fordele og ulemper kan der være ved at lade sikkerhedsadvarsler stoppe CI-pipelinen?

Del 6 – Undersøg sårbarhedens betydning

Det er ikke tilstrækkeligt kun at registrere, at en sårbarhed findes.

Vi skal også undersøge, hvad sårbarheden betyder.

Trin 1 – Find sårbarhedsbeskrivelsen

Åbn linket til den relevante sikkerhedsadvarsel.

For den bevidst inkluderede `Newtonsoft.Json`-version kan I blandt andet undersøge:

https://github.com/advisories/GHSA-5crp-9r3c-p9vr

Trin 2 – Undersøg følgende

- Hvilken type sikkerhedsproblem beskrives?
- Hvilke versioner af biblioteket er berørt?
- Findes der en rettet version?
- Hvilke forudsætninger skal være opfyldt, for at sårbarheden kan udnyttes?
- Hvordan kan applikationens brug af biblioteket påvirke den faktiske risiko?

Sikkerhedsmæssig vurdering

En sårbarhed i et bibliotek er ikke nødvendigvis ensbetydende med, at den konkrete applikation kan udnyttes.

Det afhænger blandt andet af:

- Om applikationen anvender den berørte funktionalitet.
- Om en angriber kan påvirke de relevante input.
- Om der findes begrænsninger eller andre sikkerhedsforanstaltninger.

Derfor er det vigtigt at kombinere automatiseret scanning med manuel vurdering.

I skal ikke rette eller opdatere `Newtonsoft.Json` endnu.

Vi bevarer afhængigheden, så den senere kan anvendes i øvelsen om Dependabot.

Refleksion

Hvorfor kan en automatiseret sårbarhedsscanning ikke alene afgøre, hvor stor den faktiske risiko er for den konkrete applikation?

Del 7 – Valgfri: Lad CI fejle, hvis scanningen finder sårbarheder

Indtil nu har NuGet-scanningen rapporteret kendte sårbarheder, uden at dette nødvendigvis har fået CI-pipelinen til at fejle.

I denne valgfrie del skal I ændre scannings-steppet, så det **returnerer en fejlkode, når der identificeres en eller flere kendte sårbarheder**.

Dermed undersøger vi, hvordan resultater fra automatiserede sikkerhedsværktøjer kan anvendes til at styre en CI-proces.

Trin 1 – Forstå exit codes

Når et program udføres, afsluttes processen normalt med en **exit code**.

Denne kode fortæller det omgivende miljø, hvordan processen blev afsluttet.

Exit code	Betydning
`0`	Processen blev gennemført succesfuldt
`1`	Processen rapporterede en fejl

GitHub Actions anvender processens exit code til at afgøre, om et step skal markeres som succesfuldt eller fejlet.

En NuGet-scanning kan gennemføres korrekt og returnere exit code `0`, selvom den identificerer sårbare afhængigheder.

Vi kan derfor tilføje vores egen logik, som undersøger resultatet og returnerer en fejlkode, hvis der findes sårbarheder.

Trin 2 – Opdater scanning-steppet

Åbn:

`.github/workflows/ci.yml`

Find det eksisterende step:

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

Erstat det med:

```yaml
      - name: Scan NuGet dependencies
        shell: bash
        run: |
          dotnet package list --vulnerable --include-transitive --no-restore
--format json --output-version 1 > vulnerability-report.json

          cat vulnerability-report.json

          count=$(jq '[.projects[]? | .frameworks[]? |
            (.topLevelPackages[]?, .transitivePackages[]?) |
            .vulnerabilities[]?] | length' vulnerability-report.json)

          echo "Number of vulnerability findings: $count"

          if [ "$count" -gt 0 ]; then
```

```
            echo "Vulnerable dependencies detected!"
            exit 1
          fi

          echo "No known vulnerable dependencies detected."
```

Bemærk, at dette er et komplet step, som skal placeres under `steps` i det eksisterende `build`-job.

Forklaring

`shell: bash`

Angiver, at kommandoerne skal udføres med Bash.

Vi anvender Bash, fordi workflowet kører på en Ubuntu-runner.

`run: |`

Gør det muligt at angive flere kommandoer i det samme GitHub Actions-step.

`--format json`

Angiver, at NuGet skal returnere resultatet i JSON-format.

Dette gør det muligt at behandle resultatet programmæssigt.

`--output-version 1`

Angiver versionen af NuGets JSON-outputformat.

Det gør det lettere at arbejde med en kendt datastruktur.

`--no-restore`

Undlader at udføre en ny restore, inden pakkerne undersøges.

Det er muligt, fordi afhængighederne allerede er blevet gendannet tidligere i CI-jobbet.

`> vulnerability-report.json`

Gemmer scanningens output i en JSON-fil.

`cat vulnerability-report.json`

Viser rapportens indhold i GitHub Actions-loggen.

`jq`

Er et værktøj, der kan læse og behandle JSON-data.

I dette eksempel anvendes `jq` til at tælle registrerede sårbarheder i både direkte og transitive afhængigheder.

Kommandoen:

```
count=$(jq '[.projects[]? | .frameworks[]? |
  (.topLevelPackages[]?, .transitivePackages[]?) |
  .vulnerabilities[]?] | length' vulnerability-report.json)
```

gemmer antallet af rapporterede sårbarhedsfund i variablen `count`.

`if [ "$count" -gt 0 ]`

Kontrollerer, om der er fundet mindst én sårbarhed.

`exit 1`

Afslutter steppet med en fejlkode.

GitHub Actions vil derfor markere steppet og det samlede job som fejlet.

Hvis scriptet gennemføres uden fejl og uden at finde sårbarheder, afsluttes det normalt med exit code `0`.

Trin 3 – Commit og push ændringen

Gem workflowet.

Overfør ændringen til GitHub:

```
git add .github/workflows/ci.yml
git commit -m "Fail CI when vulnerable dependencies are detected"
git push origin main
```

Åbn derefter det nyeste workflow-run under **Actions**.

Trin 4 – Analyser resultatet

Undersøg følgende:

1. Blev dependency-scanningen gennemført?
2. Hvilke sårbare afhængigheder blev identificeret?
3. Hvilken exit code returnerede steppet?
4. Hvordan blev steppet markeret i GitHub Actions?
5. Hvilken status fik det samlede CI-job?

Forventet resultat

Da projektet indeholder en bevidst sårbar NuGet-afhængighed, forventes scanningen at identificere mindst én kendt sårbarhed.

Scriptet returnerer derfor `exit 1`.

GitHub Actions markerer derefter scanning-steppet og CI-jobbet som fejlet.

Det er vigtigt at skelne mellem:

- At scanningen ikke kunne gennemføres på grund af en teknisk fejl.
- At scanningen blev gennemført og identificerede sårbarheder.
- At vores script bevidst returnerede en fejlkode på baggrund af de identificerede sårbarheder.

I det sidste tilfælde er det vores egen kontrolpolitik, som bestemmer, at CI-jobbet skal fejle.

Bemærk også, at et fejlet CI-job ikke automatisk forhindrer et push til `main`.

Dette kræver yderligere konfiguration af eksempelvis GitHubs branch protection rules eller rulesets.

Trin 5 – Gendan den oprindelige konfiguration

For at kunne fortsætte med de efterfølgende øvelser skal I gendanne det oprindelige scanning-step:

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

Gem filen, og udfør:

```
git add .github/workflows/ci.yml
git commit -m "Restore NuGet vulnerability reporting"
git push origin main
```

Kontroller, at CI-pipelinen igen gennemføres.

Vi beholder den sårbare afhængighed, så den også kan anvendes i de efterfølgende øvelser.

Refleksion

1. Hvad er forskellen på, at en scanning finder en sårbarhed, og at scanningsprocessen fejler?
2. Hvordan kan exit codes anvendes til at integrere sikkerhedskontroller i CI?

3. Hvilke fordele kan der være ved automatisk at lade CI fejle, hvis en sårbarhed identificeres?
4. Hvilke ulemper kan der være ved at lade alle sårbarheder – uanset alvorlighedsgrad – medføre et fejlet CI-job?
5. Hvordan kunne scriptet ændres, så kun sårbarheder med alvorlighedsgraden High eller Critical medfører en fejl?

Samlet refleksion

Besvar følgende spørgsmål med udgangspunkt i jeres egne observationer:

1. Hvad er forskellen mellem en direkte og en transitiv afhængighed?
2. Hvordan kan 3.-partssårbarheder introducere sikkerhedsproblemer i en applikation?
3. Hvad bidrager NuGet Audit med i forhold til de automatiserede builds og unit tests fra øvelse 38 og 39?
4. Hvorfor kan en CI-pipeline gennemføres succesfuldt, selvom den rapporterer en kendt sårbarhed?
5. Hvad er forskellen mellem at identificere en kendt sårbarhed og at verificere, at den kan udnyttes i applikationen?
6. Hvordan kan automatiseret dependency scanning indgå i en mere sikker softwareudviklingsproces?

Resultat og evaluering

Øvelsen er gennemført, når I kan demonstrere følgende:

- I har undersøgt projektets NuGet-afhængigheder.
- I har udført NuGet Audit lokalt.
- I har identificeret den bevidst sårbare `Newtonsoft.Json`-afhængighed.
- I har udvidet GitHub Actions-workflowet med scanning af sårbare afhængigheder.
- I har udført workflowet og undersøgt sikkerhedsfundene.
- I kan forklare forskellen mellem en sikkerhedsadvarsel og et fejlet CI-job.
- I kan forklare, hvorfor fund fra automatiseret scanning kræver en sikkerhedsmæssig vurdering.

Den valgfrie Del 7 giver yderligere mulighed for at demonstrere, hvordan CI-pipelinen kan konfigureres til at fejle på baggrund af scanningens resultater.

Det vigtigste er ikke blot at gennemføre scanningen, men at forstå **hvad værktøjet undersøger, hvilke oplysninger det giver, og hvordan disse oplysninger kan anvendes i softwareudviklingsprocessen**.

Videre arbejde

I øvelse 38 etablerede I en automatiseret buildproces.

I øvelse 39 tilføjede I automatiserede unit tests.

I denne øvelse har I tilføjet scanning af tredjepartsafhængigheder.

CI-pipelinen kan nu give feedback om:

1. Kompileringsfejl.
2. Fejlede unit tests.
3. Kendte sårbarheder i tredjepartsafhængigheder.

Men selvom alle afhængigheder er opdaterede, kan applikationens egen kildekode stadig indeholde sikkerhedsproblemer.

I **øvelse 41** udvider vi derfor sikkerhedsarbejdet med statisk kodeanalyse ved hjælp af **CodeQL**.

Litteratur og dokumentation

- *Secure by Design*, kapitel 8 – Leveraging your delivery pipeline for security.
- Microsoft – Auditing package dependencies for security vulnerabilities
- Microsoft – dotnet package list
- Microsoft – NU1901–NU1904 warnings
- GitHub Advisory – Newtonsoft.Json

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