

▮ Øvelse 35 – *Introduktion til automatiseret testing og unit tests*

▮ Information

Formålet med denne øvelse er at introducere dig til det overordnede koncept bag automatiseret test af implementeret kildekode samt hvordan en unit test kan anvendes til at teste kode.

Automatiseret test

En automatiseret test er en metode til at verificere, at software fungerer som forventet ved at bruge specialiserede testværktøjer eller scripts til at udføre testcases. I stedet for manuel test kan automatiserede tests hurtigt køre mange testscenarier uden menneskelig indblanding. En stor fordel er, at tests kan genkøres efter kodeændringer for at sikre, at funktionaliteten stadig virker korrekt og forhindre nye fejl (potentielt CVE'er).

Unit test

I faget software sikkerhed fokuserer vi på *unit tests*, som tester en isoleret del af koden — typisk en metode eller funktion. En unit test giver input til en kodeenhed og kontrollerer, at output eller opførsel er som forventet, f.eks. returnerer en bestemt værdi eller kaster en forventet exception.

Unit tests og sikkerhed

Princippet ligner den måde, du har testet applikationers sårbarheder via f.eks. Burp Suite i *Webapplikation sikkerhed*, men her er fokus på enkelt dele af koden. Unit tests hjælper med at forhindre, at ændringer i kode utilsigtet introducerer fejl og sårbarheder. Hvis udvikler X har skrevet kode og unit tests, kan udvikler Y bruge disse tests til at sikre, at ændringer ikke bryder eksisterende funktionalitet.

Særligt domænelogik er godt egnet til unit tests, fordi generelle automatiserede sikkerhedsanalyser (fx taint-analyse) ikke fanger domænespecifikke fejl, som f.eks. at en ordre kan være -1. En unit test kan sikre, at sådanne regler altid overholdes.

I øvelsen følger du en tutorial, hvor du opretter et klassebibliotek, der kan vurdere, om et tal er primtal, og et testprojekt med *XUnit* til unit tests. Bemærk brugen af løsning (`.sln`) til at forbinde projekterne — en vigtig praksis, som du skal bruge i næste øvelse, hvor du selv implementerer tests.

Eksempel på unit test i C

Nedenfor ses et simpelt eksempel på en klasse med en metode, der afgør, om et tal er et primtal, samt en tilhørende unit test-klasse, der anvender Xunit til at teste denne metode.

```
// Klasse der skal testes
public class NumberChecker
{
    public bool IsPrime(int number)
    {
        if (number <= 1) return false;
        for (int i = 2; i <= Math.Sqrt(number); i++)
        {
            if (number % i == 0) return false;
        }
        return true;
    }
}
```

```
// Unit test klasse med Xunit
using Xunit;

public class NumberCheckerTests
{
    [Fact]
    public void IsPrime_WithPrimeNumber_ReturnsTrue()
    {
        // Arrange
        var checker = new NumberChecker();

        // Act
        bool result = checker.IsPrime(7);

        // Assert
        Assert.True(result);
    }

    [Fact]
    public void IsPrime_WithNonPrimeNumber_ReturnsFalse()
    {
        // Arrange
        var checker = new NumberChecker();

        // Act
        bool result = checker.IsPrime(10);

        // Assert
        Assert.False(result);
    }
}
```

Forklaring

I dette eksempel har vi en simpel klasse `NumberChecker` med en metode `IsPrime`, som afgør, om et heltal er et primtal. I testklassen `NumberCheckerTests` har vi to unit tests, der validerer, at metoden returnerer korrekt resultat for henholdsvis et primtal (7) og et ikke-primtal (10).

Unit tests består typisk af tre trin:

- **Arrange:** Opsætning af testobjekt og data
- **Act:** Kald af den metode, der skal testes
- **Assert:** Kontrol af, at resultatet stemmer overens med forventningen

Dette eksempel illustrerer, hvordan man kan automatisere tests af små kodeenheder og sikre, at funktionaliteten virker som forventet – en vigtig del af Secure by Design.

Det er en god idé at notere dine erfaringer fra tutorialen til næste øvelse, hvor du selv skal skrive tests.

□ Instruktioner

1. □ Gennemfør [denne tutorial](#) som en grundlæggende introduktion til unit testing.
 2. □ Reflekter over, hvordan antallet og typen af testcases kan påvirke kvalitet og sikkerhed i software.
 3. □ Beskriv din forståelse af Test Driven Development (TDD) og hvordan det kan understøtte et Secure by Design mindset.
 4. □ Sammenlign den testtype, du har arbejdet med i tutorialen, med de fire testtyper præsenteret i Tabel 8.1 fra dagens forberedelse. Hvilke fordele og ulemper ser du ved netop denne testtype?
 5. □ Diskutér, hvorfor mange udviklingsteams undlader at anvende unit tests i praksis.
-

□ Refleksionsspørgsmål

1. □ Hvordan kan automatiserede tests som unit tests forbedre softwarekvalitet og mindske sikkerhedsrisici?
2. □ På hvilke måder kan Test Driven Development hjælpe med at skabe mere sikker software?
3. □ Hvilke kompromiser findes der mellem at skrive mange unit tests og udviklingshastighed?

4. ☐ Hvordan kan unit tests supplere andre sikkerhedsforanstaltninger som statisk analyse og manuel review?
5. ☐ Hvad kan være barrierer for udbredt brug af unit tests i udviklingsteam, og hvordan kan disse overvindes?

☐ Kobling til relevante standarder og principper

Element i øvelsen	Relateret standard / princip	Kommentar
Automatiseret test og unit tests	ISO/IEC 27002:2022, afsnit 8.29 (Security testing)	Fremhæver brugen af automatiserede tests for at sikre kontinuerlig kvalitet og sikkerhed.
	OWASP ASVS 5.0.0, kapitel 14 (Security Testing)	Krav om systematisk test af applikationer for at identificere sårbarheder tidligt.
	Secure by Design (Johnsson et al.)	Understreger TDD og automatiserede tests som nøgleelementer i sikker softwareudvikling.

☐ *Unit tests er fundamentet for robust, sikker og vedligeholdbar software, da de sikrer, at kodeændringer ikke utilsigtet skaber fejl eller sårbarheder.*

Last update: 2025-10-30 12:07:19