

Security Assessment – Attack & Defense Simulation

SQL Injection mot UC-baserat kreditbeslut

1. Inledning

Detta projekt demonstrerar hur en SQL injection-sårbarhet i ett banksystem kan utnyttjas för att manipulera ett kreditbeslut som baseras på extern UC-liktande kreditdata.

Projektet genomförs i en isolerad labbmiljö (Ubuntu VM) och består av två delar: en Attack-applikation och en Defense-applikation.

Syftet är att visa både hur en sårbarhet kan exploateras och hur den kan åtgärdas enligt etablerade säkerhetsprinciper.

2. Vald sårbarhet: SQL Injection

SQL Injection uppstår när användarinput blandas direkt in i SQL-frågor utan korrekt hantering.

I detta labb används SQL injection för att manipulera ett UC-uppslag som banken använder som underlag för sitt lånebeslut.

Sårbarheten tillhör kategorin Webbapplikationssårbarheter och är en välkänd risk inom både traditionell IT-säkerhet och ICT-riskhantering.

3. Systemöversikt

Systemet består av följande komponenter:

- Webbgränssnitt för låneansökan
- Bankapplikation som:
 - hämtar UC-liktande kreditdata från databas
 - fattar beslut baserat på kreditpoäng och belopp
- Databas (SQLite) som simulerar UC-profiler

Attack- och Defense-versionerna använder samma arkitektur men olika implementation av databasåtkomst. Se bild.

Not Secure http://10.0.2.15:5002

Låneansökan

Denna tjänst använder **extern UC-likande kreditdata** som underlag för bankens beslut.

Namn

Personnummer (SSN)

Ansökt belopp (SEK)

Skicka ansökan

Security test (endast labb):
I attack-versionen kan UC-uppslag manipuleras via SQL injection.

Exempel payload för Personnummer:
`' OR 1=1 ORDER BY score DESC --`

4. Attack – Hur sårbarheten utnyttjas

4.1 Beskrivning av attacken

I attack-versionen byggs SQL-frågan med osäker stränginterpolering.

Detta gör att angriparen kan injicera SQL-kod i fältet för personnummer (SSN).

Exempel på injektionspayload:

`' OR 1=1 ORDER BY score DESC --`

Payloaden gör att databasen returnerar den UC-profil som har högst kreditpoäng, oavsett vem som ansöker.

4.2 Steg-för-steg: Attack

1. Användaren skickar en normal låneansökan → lånet nekas
2. Angriparen injicerar SQL-payload i personnummerfältet
3. SQL-frågan manipuleras
4. Banken hämtar fel UC-profil
5. Lånet godkänns felaktigt

Resultat på låneansökan med personnummer & låg uc_score

```
amount: 250000
application_id: 10
decision: "DENIED"
name: "Issaf"
note: "This is the VULNERABLE version. Try SQL injection in ssn."
ssn_submitted: "199001011234"
uc_score_used: 320
```

Manipulerat beslut – godkänd låneansökan

Låneansökan

Denna tjänst använder **extern UC-liknande kreditdata** som underlag för bankens beslut.

Namn

Personnummer (SSN)

Ansökt belopp (SEK)

Skicka ansökan

Security test (endast labb):
I attack-versionen kan UC-uppslag manipuleras via SQL injection.
Exempel payload för Personnummer:
' OR 1=1 ORDER BY score DESC --'

Godkänt ansökan för samma person efter SQLi

The screenshot shows a browser window with two tabs. The left tab has the URL 10.0.2.15:5002/apply. The right tab has the URL 10.0.2.15:5003/apply. Both tabs show a JSON editor interface. The JSON payload in the left tab is:

```
amount: 250000
application_id: 4
decision: "APPROVED"
name: "Issaf"
note: "This is the VULNERABLE version. Try SQL injection in ssn."
ssn_submitted: "' OR 1=1 ORDER BY score DESC --"
uc_score_used: 910
```

4.3 Konsekvens

Attacken leder till:

- Manipulation av affärskritiska beslut
- Felaktig kreditbedömning
- Ekonomisk risk för banken

Detta visar tydligt hur tekniska sårbarheter kan få direkta affärskonsekvenser.

5. Defense – Hur sårbarheten åtgärdas

5.1 Skyddsåtgärder

I defense-versionen har följande åtgärder implementerats:

- Parameteriserade SQL-frågor
- Inputvalidering av personnummer
- Flagging av misstänkta ansökningar

Den centrala säkerhetsprincipen är att separera data från kod.

Exempel på parametriserade SQL-frågor från koden

```
def safe_uc_lookup(ssn: str) -> int:
    """
    FIX: parameteriserad query (separerar data från kod).
    """
    conn = db()
    cur = conn.cursor()
    row = cur.execute(
        "SELECT score FROM uc_profiles WHERE ssn = ? LIMIT 1",
        (ssn,))
    .fetchone()
    conn.close()
    return int(row["score"]) if row else 0
```

5.2 Resultat efter fix

När samma SQL injection-payload skickas till defense-versionen:

- Injektionen exekveras inte
- UC-score manipuleras inte
- Ansökan nekas
- Händelsen flaggas för granskning

Skyddad version där SQLi blockeras

The screenshot shows a browser window with two tabs open. The left tab has the URL `10.0.2.15:5002/apply` and the right tab has `10.0.2.15:5003/apply`. Both tabs display a JSON response body. The JSON data is as follows:

```
amount: 250000
application_id: 3
decision: "DENIED"
flagged_for_review: "YES"
name: "Issaf"
note: "This is the DEFENDED version. SQL injection should not affect the UC lookup."
ssn_submitted: "' OR 1=1 ORDER BY score DESC --"
uc_score_used: 0
```

6. Resultatjämförelse

Scenario	UC-score	Beslut
Normal input (Attack)	Låg	Nekad
SQL Injection (Attack)	Hög	Godkänd
SQL Injection (Defense) 0 / blockerad Nekad		

Denna jämförelse visar tydligt skillnaden mellan sårbart och säkert system.

7. Etisk reflektion

All testning har skett:

- i en isolerad VM
- mot egenutvecklade system
- utan påverkan på externa system

Projektet följer de tre etiska principerna:

- Laglighet
- Auktorisering
- Rapportering

Syftet är att förbättra säkerhet, inte att orsaka skada.

8. Slutsats

Projektet visar hur en enkel SQL injection kan utnyttjas för att manipulera ett kreditbeslut när en bank litar blint på extern data.

Genom att använda parameteriserade SQL-frågor och inputvalidering elimineras risken.

Labbet demonstrerar tydligt sambandet mellan:

- teknisk sårbarhet
- ICT-risk
- affärspåverkan