SECURITATEA PRODUSELOR PROGRAM (SPP)

T3. Securizarea Cerințelor și Designului Software

- Analiza și validarea cerințelor de Securitate.
- Design securizat și modele arhitecturale

Securitatea software trebuie integrată încă din fazele incipiente ale dezvoltării unui produs software. O proiectare sigură și o validare riguroasă a cerințelor de securitate reduc riscurile de atac și vulnerabilități critice.

Obiectivele prelegerii:

- Înțelegerea importanței cerințelor de securitate în dezvoltarea software
- Metode pentru analiza și validarea cerințelor de securitate
- Principii și modele arhitecturale pentru un design securizat

---

Analiza și Validarea Cerințelor de Securitate

Ce sunt cerințele de securitate?

Cerințele de securitate sunt specificații care definesc **măsurile de protecție** pe care trebuie să le implementeze un sistem software pentru a preveni atacurile și a proteja datele. Acestea trebuie definite în **faza de analiză** a unui proiect software și validate pe tot parcursul ciclului de dezvoltare.

Exemple de cerințe de securitate:

- **Autentificare sigură** (ex.: utilizarea MFA – Multi-Factor Authentication)
- **Controlul accesului** (ex.: utilizarea rolurilor și permisiunilor)
- **Protecția datelor** (ex.: criptarea datelor sensibile)
- **Audit și jurnalizare** (ex.: stocarea evenimentelor pentru investigații)

Cum analizăm cerințele de securitate?

Analiza cerințelor de securitate implică identificarea și evaluarea riscurilor asociate unui sistem software. Metode utilizate:

1. **Modelarea amenințărilor (Threat Modeling)**

Metodă utilizată pentru a **identifica, clasifica și prioritiza** amenințările de securitate asupra unui sistem software.

- Utilizarea metodologiilor precum **STRIDE** (Microsoft) (Spoofing (Falsificare identitate), Tampering (Modificare neautorizată), Repudiation (Negare acțiuni), Information Disclosure (Divulgare informații sensibile), Denial of Service, (Blocare serviciu), Elevation of Privilege (Escaladarea privilegiilor)).
- Examinarea punctelor de intrare și ieșire ale sistemului pentru a descoperi potențiale vulnerabilități.

  Exemplu: O aplicație bancară implementează STRIDE:

  Spoofing → Se impune autentificare multi-factor.
  Tampering → Se utilizează semnături digitale pentru mesaje.
  Repudiation → Se implementează log-uri imutabile.
  Information Disclosure → Se aplică criptarea datelor.

2. **Evaluarea riscurilor**
   - Determinarea impactului unui atac asupra sistemului.
   - Clasificarea și evaluarea riscurilor în funcție de probabilitate și impact (ex.: utilizarea **metodei DREAD** – Damage, Reproducibility, Exploitability, Affected Users, Discoverability). **Ex.** Scor total: 46/50 → Vulnerabilitate critică, necesită remediere urgentă!
3. **Metode de validare a cerințelor de securitate**
   - **Analiza formală** – Utilizarea unor modele matematice pentru verificarea securității.

- **Revizuiri manuale** – Evaluarea documentației și codului de către experți în securitate.
- **Analiză statică a codului** – Detectarea vulnerabilităților fără execuția codului.
- **Fuzzing** – Injectarea de date invalide pentru a identifica crash-uri și exploatări.
- **Testare de penetrare** – Simularea atacurilor pentru a verifica eficiența măsurilor de securitate.

Exemple:
O companie dezvoltă o aplicație bancară. În analiza cerințelor de securitate, identifică următoarele riscuri:

- Atacuri prin **phishing** asupra utilizatorilor.
- **Injection attacks** (ex.: SQL Injection) asupra bazei de date.
- Acces neautorizat la conturile utilizatorilor.

Soluție: Se implementează autentificarea MFA, validarea riguroasă a input-urilor și criptarea datelor.

Un sistem bancar implementează testare de penetrare → descoperă că input-ul pentru câmpul „username” permite SQL Injection. Soluție: utilizarea prepared statements (instrucțiuni pregătite anterior).

Design Securizat și Modele Arhitecturale

Principii ale Designului Securizat

Principiul privilegiului minim – Utilizatorii și procesele primesc doar accesul strict necesar.
Principiul separării responsabilităților – Evitarea concentrării funcționalităților critice într-un singur modul sau utilizator.
Fail-safe defaults – Implicit, accesul este restricționat, iar permisiunile sunt acordate explicit.
Defense-in-depth – Implementarea mai multor niveluri de protecție pentru a minimiza impactul unei breșe de securitate.

Modele Arhitecturale Securizate

1. Modelul Zero Trust

- **"Nu aveți încredere în nimeni, verificați totul"**
- Toți utilizatorii și dispozitivele trebuie să fie autentificate și autorizate înainte de a accesa resurse.
- Exemple de măsuri: MFA, segmentarea rețelei, criptare end-to-end, autentificare bazată pe context (verifică locația utilizatorului).

2. Modelul Secure by Design

- Securitatea este integrată încă din faza de proiectare, nu adăugată ulterior.
- Se utilizează practici precum:
    - Utilizarea bibliotecilor sigure
    - Evitarea hardcodării parolelor
    - Implementarea controalelor de acces

3. Arhitectura cu Microservicii și Securitate

- Separarea aplicației în module independente pentru a limita impactul unui atac.
- Utilizarea **API Gateway** pentru gestionarea accesului.
- Criptarea comunicării între microservicii.

4. Modelul Arhitectural **"Layered Security"** (Defense-in-Depth)

- Securitatea este implementată în **straturi multiple** pentru a proteja datele și aplicațiile.
- Exemple: firewall-uri, autentificare puternică, detecție a intruziunilor (IDS/IPS), control biometric + criptare disk.

Studii de Caz și Exemple

Exemplu 1: Proiectarea unei aplicații web securizate

- **Cerință:** Dezvoltarea unui portal pentru clienți cu acces la date financiare.
- **Analiza riscurilor:**
    - Posibile atacuri de tip **Cross-Site Scripting (XSS)** și **SQL Injection**.

- **Măsuri de securitate:**
    - Validarea input-ului utilizatorului.
    - Utilizarea **OAuth 2.0** pentru autentificare sigură.
    - Aplicarea principiului **"Least Privilege"** pentru accesul utilizatorilor.

Exemplu 2: Securizarea unei aplicații de IoT

- **Cerință:** Proiectarea unui sistem de case inteligente conectate la internet.
- **Provocări:**
    - Dispozitive IoT vulnerabile la atacuri prin rețea.
- **Soluții:**
    - Implementarea criptării TLS pentru comunicații.
    - Autentificare unică pentru fiecare dispozitiv.
    - Monitorizare continuă a traficului de rețea.

Concluzii și Bune Practici

Integrarea securității în **fiecare etapă a ciclului de viață al software-ului**.
Utilizarea **analizei amenințărilor** pentru a identifica și minimiza riscurile.
Aplicarea principiilor de **design securizat** și implementarea unor **arhitecturi robuste**.
Validarea continuă a cerințelor și a implementării măsurilor de securitate.

Securitatea nu este un proces unic, ci unul continuu. O aplicație sigură este una care evoluează constant pentru a face față noilor amenințări.

T4. Tehnici de Programare Securizată. Pratici de programare pentru prevenirea vulnerabilităților commune si exemple

Programare securizată în C/C++

Securitatea software este o preocupare majoră în prezent. Nu puteți risca nicio vulnerabilitate de securitate - în special dacă dezvoltați software pentru sisteme integrate. Iar acest lucru înseamnă că codul dvs. trebuie să fie sigur și fără erori de codare.

Când vă gândiți la securitatea software, probabil vă gândiți la parole și la controlul accesului. Sau la viruși, spoofing și atacuri de phishing.

Acestea sunt preocupări comune privind securitatea. Iar funcțiile de securitate, precum criptarea datelor și protocoalele de autentificare, atenuează aceste vulnerabilități.

Dar chiar dacă ați implementat aceste caracteristici de securitate, software-ul poate rămâne vulnerabil.

Pentru a asigura un software sigur, trebuie să începeți de la sursă - de la nivelul codului. În caz contrar, erorile de codare vă vor compromite programul.

Erorile de codificare compromit securitatea

Software Engineering Institute (SEI) estimează că până la 90% din de securitate raportate incidentele rezultă din exploatarea vulnerabilităților din codul sau din proiectarea software-ului. Iar aceste vulnerabilități permit hackerilor să acceseze date private sau să preia controlul neautorizat al unui sistem.

Astfel, o simplă eroare de codare poate duce la o amenințare de hacking. Un hacker ar putea prelua controlul asupra computerului, dispozitivului de automatizare a locuinței, dispozitivului de divertisment de acasă sau mașinii. Mai rău, un hacker ar putea prelua controlul chiar și asupra unei centrale nucleare.

EXEMPLU DE VULNERABILITATE DE SECURITATE:

BUFFER OVERFLOW

Pentru a ilustra cum se poate întâmpla acest lucru, să ne uităm la un singur exemplu. Buffer overflow este o vulnerabilitate de securitate comună în programarea C.

Ce este Buffer Overflow?

Depășirea tamponului are loc atunci când datele sunt scrise în afara limitei memoriei alocate.

De exemplu

```
char buff[10];
buff[10]= 'a';
```

Aici este declarat un array de 10 octeți (indexul 0-9). Dar programul încearcă apoi să scrie un caracter cu un octet dincolo de limita array-ului. Dacă memoria învecinată array-ului este utilizată mai târziu în program, atunci aceasta va duce la un comportament neașteptat.

Acest lucru este destul de rău. Și poate deveni și mai rău. Un buffer overflow poate permite unui hacker să preia controlul asupra
un sistem.

Cum Buffer Overflow invită hackerii.

Hackerii pot utiliza erorile de tip buffer overflow pentru a bloca un program, pentru a corupe datele sau pur și simplu pentru a fura informații.

Atunci când un program rulează, acesta utilizează o zonă de memorie denumită "stivă". Variabilele din domeniul de aplicare al funcției în curs de execuție vor fi stocate pe stivă. Adresa apelului funcției va fi, de asemenea, stocată pentru a permite instrucțiunilor de revenire să revină la locația corectă.

Atunci când funcția se întoarce la funcția apelantă, execuția programului continuă de unde a rămas. Prin urmare, dacă adresa de întoarcere de pe stivă este modificată pentru a indica unele instrucțiuni rău intenționate alternative, atunci instrucțiunile respective vor fi executate atunci când funcția se întoarce.

Dacă programul primește date - și nu există nicio verificare care să asigure că tamponul de intrare nu se poate revărsa - atunci va fi posibil să se conceapă o intrare, sau o "sarcină utilă", care conține cod malițios. Astfel, bufferul de intrare este depășit și adresa de retur de pe stivă este suprascrisă cu adresa codului malițios.

PREVENIREA VULNERABILITĂȚILOR DE SECURITATE ESTE ESENȚIALĂ

Prevenirea vulnerabilităților de securitate - cum ar fi buf- fer overflow - este

esențială. Iar acest lucru se poate realiza prin asigurarea faptului că codul în sine este scris fără lacune exploatabile.

La urma urmei, dacă puneți încuietori mai puternice la ușa de la intrare, nu are niciun rost dacă ferestrele sunt lăsate deschise. Deci, pentru a îmbunătăți securitatea, va trebui să asigurați un cod sigur.

4 moduri de a asigura un cod sigur în C

Scrierea unui cod sigur este importantă. Iar când vine vorba de programarea C, există patru surse cheie de informații care vă ajută să asigurați un cod sigur.

1. CWE

Puteți identifica deficiențele de securitate din Enumerarea deficiențelor comune (CWE).

Ce este CWE?

CWE este o listă dezvoltată de comunitate a punctelor slabe comune de securitate software în C. Este menținută de MITRE Corporation. Această listă poate fi utilizată ca bază pentru identificarea, atenuarea și prevenirea deficiențelor.

Lista CWE a deficiențelor de securitate software

Lista CWE prioritizează punctele slabe. Primele 25 de intrări sunt prioritizate pe baza informațiilor primite de la mai mult de două duzini de organizații diferite. Acestea evaluează fiecare punct slab pe baza frecvenței și importanței. Multe dintre punctele slabe (ale programelor C) enumerate în CWE se referă la debordarea de buffer.

Lista primelor 25 adaugă, de asemenea, un mic set al celor mai eficiente "Monster Mitigations". Acest lucru ajută dezvoltatorii să reducă sau să elimine grupuri întregi din primele 25 de puncte slabe. De asemenea, ajută în cazul multora dintre celelalte 800 de puncte slabe care sunt documentate în lista CWE.

CWE se concentrează pe stoparea vulnerabilităților la sursa. Acest lucru se realizează prin educarea proiectanților, programatorilor și verificatorilor cu privire la modul de eliminare a greșelilor frecvente - chiar înainte ca software-ul să fie livrat.

2. CERT C

Puteți aplica standardul de codare CERT C codului dumneavoastră.

Ce este CERT C?

Standardul de codare CERT C este publicat de Divizia CERT din cadrul Software Engineering Institute (SEI). SEI este un centru de cercetare și dezvoltare administrat de Universitatea Carnegie Mellon. Acesta este finanțat în principal de Departamentul american al Apărării și de Departamentul pentru Securitate Internă.

Reguli de securitate CERT C

Experții în coduri securizate dezvoltă în permanență liniile directoare CERT C.

Fiecare linie directoare constă :

- Un titlu
- O descriere
- Un exemplu de cod neconform
- Exemple de soluții conforme

Orientările acoperă erorile de codare și implementare, precum și erorile de proiectare de nivel scăzut. Scopul este de a elimina practicile de codare sigură și comportamentele nedefinite care pot duce la vulnerabilități.

CERT C definește o vulnerabilitate ca:

Un set de condiții care permit unui atacator să încalce o politică de securitate explicită sau implicită.

Defectul poate fi minor. Aceasta înseamnă că nu afectează performanța sau rezultatele produse de către software. Dar, cu toate acestea, poate fi exploatat de un atac. Iar acest lucru duce la o încălcare semnificativă a securității.

3. ISO/IEC TS 17961:2013 "C SECURE"

Puteți aplica regulile de codificare ISO/IEC TS 17961:2013 "C Secure".

Ce este ISO/IEC TS 17961:2013?

ISO/IEC TS 17961:2013 stabilește un set de reguli de codare. Aceste reguli permit analizoarelor statice de cod să diagnosticheze codul nesigur dincolo de cerințele standardului de limbaj.

C Reguli de codificare sigură

ISO/IEC TS 179671:2013 include reguli pentru codarea sigură în C. De asemenea, include exemple pentru fiecare regulă.

Scopul C Secure este de a specifica reguli de codare sigură care pot fi

aplicate automat. Acestea pot fi utilizate pentru a detecta deficiențele de securitate în programarea C. Pentru a fi considerat un defect de securitate, un defect software trebuie să poată fi declanșat de acțiunile unui utilizator sau atacator rău intenționat.

Analizoarele care implementează aceste reguli trebuie să fie capabile să descopere în mod eficient erori de codare securizată - fără a genera pozitivi falși excesivi.

4. MISRA C

De asemenea, puteți utiliza MISRA pentru a asigura codificarea sigură în C.

Ce este MISRA?

MISRA oferă orientări privind cele mai bune practici pentru dezvoltare a sistemelor legate de siguranță.
Programul său C standardele de codificare au fost adoptate pe scară largă în multe industrii.

Reguli de securitate MISRA C

MISRA C:2012 Amendamentul 1 a fost publicat în 2016. Acesta oferă orientări suplimentare privind securitatea pentru programarea C, inclusiv reguli și directive noi. De asemenea, include exemple de coduri conforme și neconforme.

Aceste orientări pot fi utilizate pentru a preveni erorile de codare care conduc la probleme de siguranță și vulnerabilități de securitate.

De ce regulile de securitate MISRA C sunt ideale pentru sistemele integrate

Normele de securitate MISRA C sunt ideale pentru sistemele integrate. Acest lucru se datorează faptului că securitatea MISRA C este egală cu cea a altor standarde de codare sigură pentru C. În plus, MISRA C este recunoscut în toate industriile de sisteme integrate. Și este un standard de codificare utilizat în industria auto.

EXEMPLU DE REGULĂ DE SECURITATE MISRA C

Regulile de securitate MISRA C pot preveni erorile de codare și deficiențele de securitate, cum ar fi debordarea de buffer.

Iată un exemplu de regulă de securitate MISRA C:

MISRA C Regula 18.1

"Un pointer rezultat în urma aritmeticii asupra unui operand de pointer se adresează unui element al aceluiași array ca și operandul de pointer."

Această regulă face același lucru ca și următoarea regulă CERT C.

> ARR30-C
>
> "Nu formați și nu utilizați pointeri în afara limitelor sau subscripții de matrice."

Și ambele se referă la multiple puncte slabe ale CWE în C, dintre care una este:

> CWE-119: Restricționarea necorespunzătoare a operațiunilor în limitele unui tampon de memorie
>
> "Software-ul efectuează operații pe un buffer de memorie, dar poate citi din sau scrie într-o locație de memorie care se află în afara limitei prevăzute a zonei tampon."

Respectarea regulii MISRA C sau a regulii CERT va asigura un cod sigur și va evita deficiențele comune ale CWE. Acest lucru se datorează faptului că scrierea la un pointer (sau operand de pointer) în afara intervalului ar putea duce la o depășire a bufferului - și la un cod vulnerabil. Citirea de la un pointer (sau operand de pointer) în afara domeniului ar putea dezvălui accidental informații hackerilor.

Astfel, asigurându-vă că aceste reguli sunt respectate, veți evita erori grave de codare.

Exemple practice

Evitarea utilizării funcțiilor nesigure – ex. gets(), strcpy(), sprintf(), scanf()

Utilizarea funcțiilor sigure – ex. strncpy(), snprintf(), fgets()

Validarea strictă a datelor de intrare

Gestionarea corectă a memoriei – evitarea buffer overflow, use-after-free, double free

Evitarea codului dependent de platformă și compresia la standarde moderne (C11, C++17/20)

Utilizarea mecanismelor de protecție oferite de compilator

1. Buffer Overflow (Stack și Heap)

Descriere: O variabilă depășește memoria alocată și suprascrie date adiacente.

Stack-Based Buffer Overflow

Cod vulnerabil:

```
#include <stdio.h>
#include <string.h>
void bufferOverflow() {
    char buffer[10];
    strcpy(buffer, "Aceasta este o depasire de buffer!"); // Depășire
    printf("%s\n", buffer);
}
int main() {
    bufferOverflow();
    return 0;
}
```

Soluție: Folosirea strncpy() și fgets().

```
#include <stdio.h>
#include <string.h>
void safeBuffer() {
    char buffer[10];
    strncpy(buffer, "Sigur", sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    printf("%s\n", buffer);
}
int main() {
    safeBuffer();
    return 0; }
```

Heap-Based Buffer Overflow

Cod vulnerabil:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void heapOverflow() {
    char *buffer = (char*)malloc(10);
    strcpy(buffer, "Aceasta depaseste bufferul heap!"); // Suprascriere memorie heap
    printf("%s\n", buffer);
    free(buffer);
}
int main() {
    heapOverflow();
    return 0;
}
```

Soluție: Folosirea snprintf() și validarea lungimii.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void safeHeap() {
    char *buffer = (char*)malloc(10);
    snprintf(buffer, 10, "Sigur");
    printf("%s\n", buffer);
    free(buffer);
}
```

```
int main() {
    safeHeap();
    return 0;
}
```

2. Use-After-Free și Double Free

Use-After-Free

Cod vulnerabil:

```
#include <stdio.h>
#include <stdlib.h>
void useAfterFree() {
char *ptr = (char*)malloc(10);
    free(ptr);
    printf("%s\n", ptr); // Acces ilegal după free
}
int main() {
    useAfterFree();
    return 0;
}
```

Soluție: Setarea pointerului la NULL după free().

```
#include <stdio.h>
#include <stdlib.h>
void safeMemory() {
    char *ptr = (char*)malloc(10);
    if (ptr) {
```

```
        strcpy(ptr, "Test");
        printf("%s\n", ptr);
    }
    free(ptr);
    ptr = NULL;
}

int main() {
    safeMemory();
    return 0;
}
```

Double Free

Cod vulnerabil:

```
#include <stdio.h>
#include <stdlib.h>
void doubleFree() {
    char *ptr = (char*)malloc(10);
    free(ptr);
    free(ptr); // A doua eliberare duce la coruperea memoriei
}
int main() {
    doubleFree();
    return 0;
}
```

Soluție: După free(), setăm pointerul la NULL.

```
#include <stdio.h>
#include <stdlib.h>
void safeDoubleFree() {
    char *ptr = (char*)malloc(10);
    free(ptr);
    ptr = NULL;
    free(ptr); // Nu va avea efect
}
int main() {
    safeDoubleFree();
    return 0;
}
```

3. Integer Overflow și Truncation

Integer Overflow

Cod vulnerabil:

```
#include <stdio.h>
#include <limits.h>
void integerOverflow() {
    int x = INT_MAX;
    x += 1; // Depășire de capacitate
    printf("x: %d\n", x);
}
```

```
int main() {
    integerOverflow();
    return 0;
}
```

Soluție: Folosirea __builtin_add_overflow().

```
#include <stdio.h>
#include <limits.h>

void safeInteger() {
    int x = INT_MAX, result;
    if (__builtin_add_overflow(x, 1, &result)) {
        printf("Depășire de capacitate!\n");
    } else {
        printf("Rezultat: %d\n", result);
    }
}

int main() {
    safeInteger();
    return 0;
}
```

4. Format String Attack

Cod vulnerabil:

```
#include <stdio.h>


void formatString() {
    char userInput[50];
    gets(userInput);
    printf(userInput); // Poate expune memoria
}
int main() {
    formatString();
    return 0;
}
```

Soluție: Folosirea printf("%s", userInput).

```
#include <stdio.h>


void safeFormatString() {
    char userInput[50];
    fgets(userInput, sizeof(userInput), stdin);
    printf("%s", userInput);
}
```

```
int main() {
    safeFormatString();
    return 0;
}
```

5. Race Conditions (TOCTOU – Time of Check to Time of Use)

Cod vulnerabil:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

void toctouVulnerable() {
    struct stat statbuf;
    if (stat("file.txt", &statbuf) == 0) {
        // Între verificare și utilizare, fișierul poate fi modificat!
        FILE *file = fopen("file.txt", "w");
        fprintf(file, "Date nesigure!\n");
        fclose(file);
    }
}

int main() {
    toctouVulnerable();
    return 0;
}
```

Soluție: Utilizarea O_CREAT | O_EXCL în open().

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

void toctouSafe() {
    int fd = open("file.txt", O_CREAT | O_EXCL | O_WRONLY, 0644);
    if (fd != -1) {
        write(fd, "Date sigure!\n", 13);
        close(fd);
    } else {
        printf("Fișierul deja există!\n");
    }
}
int main() {
    toctouSafe();
    return 0;
}
```

Concluzii și Recomandări

Folosește funcții sigure: snprintf(), fgets(), strncpy()

Verifică limitele și alocarea memoriei – Evită buffer overflow

Nu folosi free() de două ori pe același pointer

Sincronizează accesul în programele multi-thread

Activează protecțiile compilatorului:

gcc -fstack-protector -D_FORTIFY_SOURCE=2 -Wl,-z,relro,-z,now

Folosește instrumente de analiză:

AddressSanitizer: gcc -fsanitize=address

Valgrind: valgrind --leak-check=full

Programare securizată în Java

Java elimină multe probleme de memorie prezente în C/C++, dar încă are vulnerabilități precum **injection attacks**, **data leakage** sau **serialization vulnerabilities**.

1. Evitarea SQL Injection

SQL Injection apare atunci când un atacator introduce cod SQL în interogările nesecurizate.

✕ Cod vulnerabil:

```
java


import java.sql.*;


public class VulnerableSQL {
    public static void main(String[] args) throws Exception {
        Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/db", "user", "password");
        Statement stmt = conn.createStatement();
        String userInput = "admin' --"; // Input rău intenționat
        String query = "SELECT * FROM users WHERE username = '" + userInput +
"'";
        ResultSet rs = stmt.executeQuery(query);
    }
}
```

✓ Cod securizat:

java

```
import java.sql.*;


public class SecureSQL {
    public static void main(String[] args) throws Exception {
        Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost/db", "user", "password");
        String query = "SELECT * FROM users WHERE username = ?";
        PreparedStatement pstmt = conn.prepareStatement(query);
        pstmt.setString(1, "admin"); // Protejează împotriva SQL Injection
        ResultSet rs = pstmt.executeQuery();
    }
}
```

2.2. Evitarea expunerii parolelor în cod

Parolele nu trebuie hardcodate în cod sursă.

✕ Cod vulnerabil:

java

```
public class InsecureAuth {
    private static final String PASSWORD = "SuperSecret123";
// Evită parole  hardcodate
}
```

✓ Cod securizat:

```
java

import java.util.Scanner;


public class SecureAuth {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter password: ");
        String password = scanner.nextLine(); // Se obține parola din input
    }
}
```

2.3. Evitarea vulnerabilităților XSS (Cross-Site Scripting)

Dacă aplicația Java interacționează cu un site web, trebuie să filtreze inputul utilizatorului pentru a preveni XSS.

✕ Cod vulnerabil:

```
java

public class InsecureWeb {
    public static String getUserComment(String input) {
        return "<p>" + input + "</p>"; // Poate introduce scripturi
malițioase
    }
}
```

✓ Cod securizat:

java

```
import org.apache.commons.text.StringEscapeUtils;

public class SecureWeb {
    public static String getUserComment(String input) {
        return "<p>" + StringEscapeUtils.escapeHtml4(input) + "</p>";
// Escape input
    }
}
```