Lucrare practica / Lucrare de laborator nr. 1

Tema: Utilizarea instrumentelor de analiză statică pentru evaluarea vulnerabilităților produselor program

Introducere

Importanța analizei statice a codului sursă: Analiza statică a codului reprezintă examinarea programelor *fără a le executa,* de obicei cu ajutorul unor unelte software automate. Aceasta a devenit parte integrantă a ciclului de viață al dezvoltării software și unul dintre primii pași în identificarea și remedierea timpurie a erorilor și vulnerabilităților din cod. Prin detectarea problemelor de securitate încă din faza de dezvoltare, se pot reduce costurile de reparare și riscul ca vulnerabilitățile să ajungă în producție. De asemenea, instrumentele automate pot analiza volume mari de cod mult mai rapid decât o pot face oamenii, examinând 100% din baza de cod și semnalând probleme critice (de ex. injection, buffer overflow etc.) cu viteză și consistență.

Diferențe între analiza statică și analiza dinamică: Analiza statică (SAST) este o metodă de testare *white-box*, ceea ce înseamnă că are acces la codul sursă al aplicației. Aceasta inspectează codul pentru a identifica puncte slabe și vulnerabilități (inclusiv cele din lista OWASP Top 10) fără a rula efectiv aplicația. Astfel, dezvoltatorii pot descoperi și remedia vulnerabilitățile în etapele timpurii ale SDLC, înainte ca aplicația să fie lansată. În schimb, analiza dinamică (DAST) este o metodă *black-box* ce presupune testarea aplicației în execuție, simulând atacuri reale din perspectiva unui hacker (fără cunoașterea codului sursă). DAST examinează comportamentul runtime al aplicației, efectuând atacuri simulate (SQLi, XSS, etc.) asupra unei instanțe live pentru a observa dacă este vulnerabilă. Pe scurt, SAST identifică vulnerabilități *mai devreme* (în cod, înainte de rulare) și la un cost mai mic de remediere, pe când DAST poate descoperi probleme care apar numai la rulare, *mai târziu* în ciclu, eventual în mediu de producție. Ideal, cele două abordări se completează reciproc într-un proces de securizare robust.

Vulnerabilități comune detectate prin analiză statică: Analiza statică se concentrează pe identificarea *slăbiciunilor cunoscute* în cod, care pot duce la vulnerabilități de securitate. Multe vulnerabilități critice își au originea în tipare de cod nesigure bine-cunoscute. De exemplu, un instrument SAST poate detecta automat probleme precum **buffer overflow** (depășirea bufferului), **SQL Injection**, **Cross-Site Scripting (XSS)**, precum și altele (injecții de comenzi, traversal de director, utilizarea funcțiilor criptografice nesigure, ș.a.). Aceste vulnerabilități apar adesea din cauza unor **pattern-uri periculoase** în cod: de pildă, concatenarea input-ului nesecurizat al utilizatorului într-o interogare SQL (favorizând SQLi), copierea datelor într-un

buffer fix fără verificarea limitelor (favorizând buffer overflow) sau afișarea ne-escape-uită a datelor introduse de utilizator în pagină (favorizând XSS). Prin detectarea automată a acestor slăbiciuni (Common Weakness Enumerations - CWE) în codul sursă sau în binare înainte de rulare, analiza statică ajută la prevenirea introducerii vulnerabilităților în produsul final

---

Analiza statică manuală

Analiza statică *manuală* presupune examinarea codului sursă de către dezvoltatori sau auditori umani, fără ajutorul direct al unui instrument automat (sau folosind doar unelte simple de navigare prin cod). Scopul este identificarea *pattern-urilor periculoase* și a erorilor de programare ce pot compromite securitatea. Această activitate este practic un **code review** focalizat pe securitate. În continuare, vom descrie metode și exemple pentru analiza manuală:

- **Metode de analiză manuală a codului sursă:** În primul rând, este esențială o **înțelegere aprofundată a funcționalității** modulului de cod analizat. Auditorul va parcurge codul linie cu linie sau folosind o abordare pe componente, căutând orice construcție care ar putea duce la o vulnerabilitate. Se folosesc liste de verificare (checklists) cu bune practici de securitate și cunoștințe despre vulnerabilitățile comune. De exemplu, se verifică dacă sunt validate toate datele de intrare, dacă sunt tratate erorile, dacă sunt utilizate funcții sigure în locul celor cunoscute ca nesigure, etc. Uneori se apelează și la căutări manuale în cod (ex: grep după funcții periculoase ca `strcpy`, `gets`, `system`, sau după construirea de stringuri SQL) pentru a marca zonele care necesită atenție. Metode complementare includ *walkthroughs* (parcurgerea codului în echipă, explicând logică) și *pair programming* focalizat pe securitate, care pot dezvălui probleme logice subtile.
- **Identificarea pattern-urilor periculoase:** Auditorul manual va fi atent la anumite *indicii* în cod care de obicei semnalează un risc de securitate. Iată câteva pattern-uri comune:
    - **Utilizarea input-ului nesanitizat:** Orice loc unde datele provenite de la utilizator (sau din exterior) sunt folosite direct, fără filtrare sau validare, poate fi problematic. Se caută de exemplu formarea de interogări SQL sau comenzi shell prin concatenarea stringurilor care includ variabile externe.
    - **Funcții nesigure sau depășite:** În limbaje precum C/C++, folosirea funcțiilor cunoscute ca nesigure (ex: `gets`, `strcpy`, `sprintf` fără limită, `scanf` fără specificator de lungime) indică posibilitatea unui overflow. Auditorul va căuta și

magical numbers sau limite hardcodate, verificând dacă există riscul de a le depăși.

- **Tratarea erorilor și a execuției neprevăzute:** Lipsa verificării valorilor de întoarcere (return codes) de la apeluri critice, manipularea necorespunzătoare a memoriei (free/double free), sau logica de autentificare/autorizare care poate fi eludată (de ex. ramuri de cod care sar peste verificarea permisiunilor) sunt și ele pattern-uri periculoase.
- **Comentarii și cod mort:** Deși nu sunt vulnerabilități în sine, prezența codului *mort* sau a secțiunilor comentate care conțin date sensibile (chei, parole) pot dezvălui informații critice. Manual, se pot identifica astfel de probleme pentru a fi eliminate.

- **Exemple de vulnerabilități frecvente și cum le recunoaștem:**
  – *SQL Injection:* apare de obicei când o interogare SQL este construită prin concatenarea stringurilor, incluzând input de la utilizator fără validare. De exemplu, dacă în cod vedem:

```
java
String user = request.getParameter("username");
String sql = "SELECT * FROM Users WHERE name = '" + user + "'";
stmt.executeQuery(sql);
```

  Auditorul recunoaște imediat pericolul: dacă `user` conține ghilimele sau operatori SQL malitioși (`' OR '1'='1` etc), interogarea devine periculoasă. **Indicatorul** este construirea manuală a stringului SQL. Soluția ar fi folosirea de parametri preparați (prepared statements) sau proceduri stocate. În analiză manuală, astfel de linii sunt marcate ca potențial *SQL injection*. (Notă: Vom vedea ulterior că instrumentele automate pot detecta aceste cazuri urmărind fluxul de date nesigure către interogări.)

  – *Buffer Overflow:* asociat în special cu limbajele ne-managementate (C/C++). Un exemplu clasic pe care un reviewer îl va detecta:

```
c
char buf[8];
gets(buf);
printf("Input: %s\n", buf);
```

  Funcția `gets` este periculoasă deoarece **nu verifică limitele** bufferului – aici, dacă utilizatorul introduce un șir mai lung de 7 caractere, se va suprascrie memorie adiacenta

lui `buf`. Un auditor de cod va remarca imediat apelul `gets` ca vulnerabilitate de **buffer overflow**, deoarece nu există nicio verificare a lungimii și documentația indică faptul că `gets` nu ar trebui folosit. Pattern-ul periculos este *oricare operațiune de copiere/lectură în buffer fix fără control al limitelor*. Soluția ar fi utilizarea unor alternative sigure (`fgets`, `gets_s`, `strncpy` cu limită, etc.) și validarea lungimii input-ului.

– *Cross-Site Scripting (XSS):* apare în aplicațiile web când datele introduse de un utilizator malițios sunt returnate către browser nesanitizate, permițând inserarea de cod HTML/JavaScript malevol. La nivel de cod, un reviewer caută locuri unde se preia parametri (de ex. din query string sau formular) și apoi se afișează direct în pagină. Exemplu simplu în PHP:

```
php
$name = $_GET['name'];
echo "<p>Bun venit, $name!</p>";
```

Dacă parametrul `name` conține `<script>alert('XSS')</script>`, acel script va fi executat de browser – deci codul de mai sus e vulnerabil. Un auditor recunoaște tiparul "*echo al input-ului direct în HTML*" ca posibil XSS, mai ales dacă nu vede nicio funcție de *escape* (precum `htmlspecialchars` în PHP sau echivalentul în alte limbaje). Ca remediu, se recomandă întotdeauna igienizarea (HTML-encoding) a output-ului sau utilizarea unor template engines care fac automat escaping.

Aceste exemple ilustrează cum, în analiza manuală, identificarea vulnerabilităților se bazează pe *experiență și cunoașterea pattern-urilor nesigure*. Este util ca evaluatorii să aibă la dispoziție și o listă de **CWE-uri** relevante și ghiduri de secure coding, pentru a verifica sistematic fiecare potențială problemă.

- **Exerciții practice pe cod sursă:** O abordare eficientă în laborator este să se exerseze analiza manuală pe fragmente de cod vulnerabil cunoscut (pentru ca studenții să se familiarizeze cu procesul):
    - Se poate furniza un program simplu (ex: o mică aplicație console C sau un script Python) care **conține intenționat câteva vulnerabilități**. Studenții, separat sau în echipe, vor parcurge codul și vor nota toate suspiciunile de vulnerabilitate găsite manual.
    - De exemplu, un program C care citește date într-un buffer fix, execută o comandă shell folosind `system()` cu input direct, sau un mic API web (pseudo-cod) care

formează interogări SQL fără sanitizare. Sarcina este ca, pas cu pas, să identifice liniile problematice și să explice ce vulnerabilitate ar putea rezulta.

- Pentru fiecare problemă găsită, se discută în grup care ar fi consecințele (ce poate face un atacator exploatând-o) și cum s-ar putea **remedia**. Această discuție face legătura între analiza statică manuala și măsurile de *secure coding*.
- Ca verificare, după ce echipele își prezintă concluziile, se poate rula pe acel cod și un instrument automat (dintre cele prezentate mai jos) pentru a vedea dacă detectează aceleași probleme – oferind o tranziție către secțiunea de analiză statică automatizată.

În concluzie, analiza statică manuală este un proces meticulos care necesită cunoștințe și concentrare, dar are avantajul că **poate descoperi atât bug-uri de securitate evidente cât și probleme logice complexe** pe care uneltele automate le pot rata. De obicei, se folosește în combinație cu instrumente automatizate: rezultatele tool-urilor sunt verificate manual (pentru a elimina false positive) și, invers, zonele sensibile identificate manual pot fi căutate și cu scanere automate pentru consistență.

Analiza statică automatizată

Analiza statică automatizată utilizează **instrumente software dedicate (scanere de securitate)** care inspectează codul sursă sau binarele și raportează potențiale vulnerabilități. Aceste instrumente pot fi integrate în procesul de dezvoltare (de ex. în pipeline-ul de CI/CD) pentru a oferi feedback rapid dezvoltatorilor. Vom prezenta în continuare câteva instrumente open-source populare pentru analiza statică, modul de instalare și utilizare a fiecăruia, precum și interpretarea rezultatelor lor:

SonarQube

Prezentare: *SonarQube* este o platformă extensibilă de analiză statică și asigurare a calității codului, care suportă multiple limbaje de programare. Aceasta examinează codul sursă pentru a detecta **defecte**, **vulnerabilități**, probleme de stil (code smells) și alte aspecte de calitate a codului. SonarQube oferă o interfață web unde dezvoltatorii pot vizualiza rapoarte detaliate despre problemele găsite și pot urmări metrice de calitate (complexitate, acoperire cu teste, duplicare de cod etc.). În contextul securității, SonarQube clasifică alertele de securitate în categorii precum *Bugs, Vulnerabilities* și *Security Hotspots*, evidențiind severitatea ("Blocker", "Critical", etc.) și oferind explicații/remedieri. (Notă: Regulile de securitate mai avansate pentru detectarea vulnerabilităților precum injecțiile pot necesita activarea *rule-set*-urilor disponibile în edițiile comerciale SonarQube (Developer, Enterprise, Data Center) cu funcționalități extinse

(analiză a codului în timp real, integrare CI/CD avansată etc.), însă pentru laborator ne vom limita la ce oferă ediția Community-comunitară).

Instalare și configurare: SonarQube are o arhitectură server-client. Instalarea implică două componente:

1. **Serverul SonarQube:** Descărcați pachetul **SonarQube Community Edition** de pe site-ul oficial SonarSource. Este necesar Java (JDK 11+). Dezarhivați pachetul și lansați serverul – pe Windows rulați `StartSonar.bat` (din directorul `bin/windows-x86-64`), iar pe Linux/Mac rulați scriptul `sonar.sh start` (din `bin/linux-x86-64` etc.). Serverul va porni pe adresa implicită `http://localhost:9000`. (La prima pornire, credențialele implicite sunt admin/admin pentru interfața web).
2. **SonarScanner (clientul de analiză):** Pentru a analiza un proiect, instalați SonarScanner CLI (Command line interface)– disponibil tot pe site-ul SonarQube. Alternativ, există integrări ca plugin Maven, Gradle, etc., dar în laborator CLI-ul este suficient. Dezarhivați SonarScanner și adăugați scriptul său în PATH pentru ușurință. În proiectul ce va fi scanat, creați fișierul de configurare `sonar-project.properties` unde specificați cel puțin:
   - `sonar.projectKey` (un identificator unic al proiectului – ex: ***LabSecuCod***),
   - `sonar.projectName` (nume descriptiv),
   - `sonar.sources` (calea către directoarele cu cod sursă),
   - eventual `sonar.language` (dacă nu autodetectează) și
   - `sonar.sourceEncoding=UTF-8`.

     Exemplu de fișier minimal:

```
ini
sonar.projectKey=LabSecuCod
sonar.projectName=Laborator Securitate Cod
sonar.sources=src
```

Utilizare (rularea analizei): După configurare, se rulează scanarea statică propriu-zisă:

1. Asigurați-vă că serverul SonarQube este pornit (`localhost:9000` disponibil). Autentificați-vă în interfața web și creați un proiect nou cu key-ul ales (dacă interfața o cere, altfel project-ul va fi creat automat la primirea primului raport).
2. Din directorul proiectului, rulați **SonarScanner**. Dacă aveți fisierul de proprietăți configurat, este suficient să executați comanda:

```
bash
sonar-scanner
```

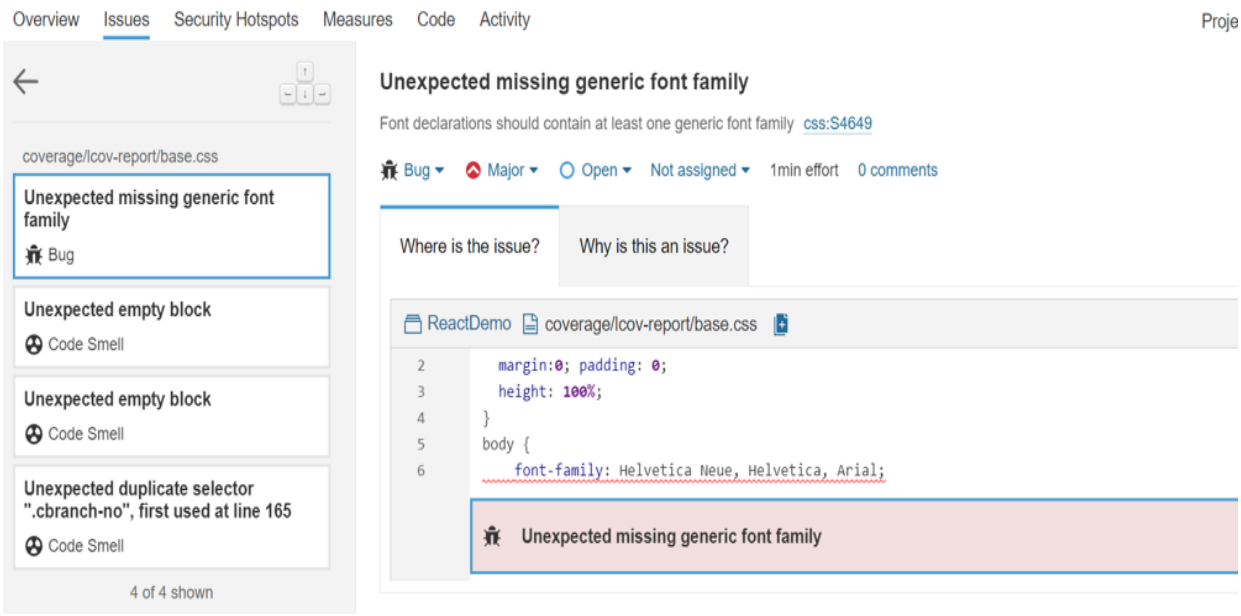
SonarScanner va citi proprietățile, va analiza codul sursă (parcurgând fișierele specificate) și va trimite raportul către serverul SonarQube. În consola locală veți vedea progresul și la final un mesaj de succes sau erori (de ex. dacă nu se poate conecta la server).

3. Odată analiza terminată cu succes, accesați interfața web SonarQube. Selectați proiectul dvs. pentru a vedea **Tabloul de bord** al analizei. Aici se afișează un sumar cu numărul de **Bugs**, **Vulnerabilities**, **Code Smells**, scorul de securitate, și alți indicatori.

Interpretarea rezultatelor: În dashboard-ul SonarQube puteți naviga la secțiunea **Issues** (Probleme) pentru a vedea lista tuturor problemelor detectate. Acestea sunt filtrabile după tip (Bug, Vulnerability, Code Smell) și severitate. Pentru fiecare problemă, SonarQube oferă:

- localizarea exactă (fișier și linie de cod) – cu un snippet de cod aferent în care este evidențiată porțiunea problematică,
- o descriere a problemei și de ce este importantă,
- un *guidance* de remediere (adesea sugestii pentru a corecta codul).

De exemplu, dacă SonarQube detectează o posibilă vulnerabilitate de **injection**, va marca linia unde input-ul nesigur este folosit și va recomanda utilizarea unui alt API sau validare. Problemele de securitate critice apar și în tab-ul **Security Hotspots** (puncte fierbinți de securitate) – aici dezvoltatorul trebuie să confirme manual dacă reprezintă vulnerabilități reale. În contextul laboratorului, vom interpreta aceste rezultate discutând de ce SonarQube a considerat o anumită construcție ca fiind vulnerabilă și dacă e un *false positive* sau ceva ce trebuie remediat.



Exemplu: Interfața web SonarQube – secțiunea "Issues" – afișând probleme detectate într-un proiect (bug-uri și code smells, cu detalii despre locație și descriere).
(Notă: În ediția comunitară, SonarQube detectează automat vulnerabilități precum injecții doar pentru anumite limbaje populare (Java, C#, PHP, JavaScript). Pentru altele, poate marca potențiale probleme drept "Security Hotspot" care necesită revizuire manuală. Pentru o acoperire mai largă a regulilor de securitate, SonarQube oferă extensii comerciale.)

Coverity Scan

Coverity Scan găsește și remediază gratuit defectele din proiectul open source Java, C/C++, C#, JavaScript, Ruby sau Python (https://scan.coverity.com). Testează fiecare linie de cod și calea potențială de execuție, iar cauza principală a fiecărui defect este explicată în mod clar, facilitând remedierea erorilor.

OWASP Dependency-Check

Prezentare: *OWASP Dependency-Check* este un instrument open-source de tip **Software Composition Analysis (SCA)**, care scanează dependențele unui proiect software (biblioteci externe, pachete) pentru a identifica vulnerabilități cunoscute în acestea. Practic, Dependency-Check compară fiecare librărie utilizată (identificată după nume, versiune, hash etc.) cu o bază de date de vulnerabilități publice (ex. CVE – Common Vulnerabilities and Exposures). Vulnerabilitățile legate de componente terțe reprezintă una din top 10 probleme de securitate conform OWASP, iar acest instrument ajută la detectarea lor automată. El poate fi folosit stand-alone (CLI) sau integrat ca plugin în build tools (Maven, Gradle) și servere CI (Jenkins, Azure

DevOps etc.). Raportul generat indică pentru fiecare dependență dacă există vulnerabilități raportate, cu detalii precum severitatea (scor CVSS), link către descrierea CVE și versiuni fixe.
Instalare: OWASP Dependency-Check are mai multe variante. Cea mai simplă în laborator este folosirea *CLI standalone*. Pașii sunt:

1. Descărcați de pe site-ul oficial OWASP pachetul **Dependency-Check CLI** (un fișier ZIP care conține un executabil `.bat` pentru Windows și script `.sh` pentru Linux/Mac, plus alte fișiere necesare).
2. *(Opțional)* Configurați variabilele de mediu dacă doriți (de ex. `PATH` către folderul `bin` al utilitarului) sau pregătiți comanda de execuție completă. De asemenea, la prima rulare, tool-ul va descărca baza de date cu vulnerabilități (de pe NVD – National Vulnerability Database); este bine să aveți conexiune la internet și poate dura câteva minute la prima actualizare.
3. Dacă proiectul vostru folosește un sistem de build (Maven/Gradle/npm), puteți folosi și pluginurile dedicate, dar în acest ghid vom exemplifica folosirea CLI generică.

Utilizare (scanarea dependențelor): Pentru a scana un proiect:

1. Executați scriptul CLI al Dependency-Check specific platformei. Comanda generală necesită specificarea unei *căi de scanare* și a unui *output* de raport. De exemplu, din directorul rădăcină al proiectului, rulați:
   ```
   bash
   dependency-check.sh --project "NumeProiect" --scan . --format HTML --
   out raport-dc.html
   ```
   (Pe Windows, folosiți `dependency-check.bat` cu aceiași parametri.)
   Acest exemplu scanează directorul curent (`--scan .`) pentru dependențe (de obicei analizează fișiere precum `pom.xml`, `package-lock.json`, `packages.config` etc., în funcție de ecosistem), atribuind proiectului un nume "NumeProiect" în raport, și generează un raport în format HTML.
2. Așteptați finalizarea scanării. La prima rulare, tool-ul va descărca și actualiza baza de date de CVE-uri, apoi va analiza dependențele. În consolă veți vedea progresul (de ex. "Checking dependencies for vulnerabilities...").
3. După terminare, deschideți fișierul HTML de raport (ex. `raport-dc.html`) cu un browser.

Interpretarea raportului: Raportul HTML al Dependency-Check conține:

- O secțiune de **rezumat**: numărul total de dependențe identificate și câte dintre ele sunt vulnerabile (clasificate pe niveluri de severitate: Critical/High/Medium/Low).
- Lista **dependințelor vulnerabile**: pentru fiecare librărie (identificată prin nume și versiune) găsită cu vulnerabilități, se listează CVE-urile relevante. Fiecare CVE are asociat un scor CVSS (gravitatea, de la 0 la 10), un link către baza de date (unde se poate citi descrierea completă a vulnerabilității) și un scurt rezumat.
- De exemplu, dacă proiectul folosește o versiune veche de biblioteca X care are o vulnerabilitate de tip deserializare nesigură, raportul poate afișa: *CVE-2023-12345 – Deserialization of untrusted data – Score 9.8 (Critical),* afectând versiuni <=1.2.3; recomandarea ar fi actualizarea la >=1.2.4.
- **False positives vs. False negatives:** Dependency-Check face o *best-effort analysis,* deci este posibil să semnaleze vulnerabilități care nu afectează efectiv aplicația (de exemplu, o librărie prezentă dar neutilizată poate figura ca vulnerabilă – se pot marca astfel de cazuri ca false positive printr-un fișier de suprimit). De asemenea, dacă dependențele nu sunt bine specificate, unele vulnerabilități pot fi ratate (false negative). În general însă, tool-ul oferă un bun punct de pornire pentru a ști ce componente trebuie actualizate.

În laborator, folosind Dependency-Check vom putea identifica dacă proiectele noastre au **componente depășite sau vulnerabile**, ilustrând importanța de a menține la zi librăriile terțe ca parte din securitatea aplicației. Remedierea constă, evident, în actualizarea acelor dependențe la versiuni fără vulnerabilități cunoscute.

Bandit (Python)

Prezentare: *Bandit* este un instrument de analiză statică dedicat codului Python, dezvoltat inițial în cadrul OpenStack Security. Scopul lui este să găsească **probleme uzuale de securitate în codul Python** prin parcurgerea abstract syntax tree (AST) al fișierelor sursă și aplicarea unui set de reguli predefinite. Bandit poate detecta, printre altele: utilizarea funcției `eval` sau `exec` cu input nesigur, string-uri SQL construite susceptibile la injection, parole hardcodate, permisiuni la fișiere incorecte, folosirea modulului `subprocess` fără măsuri de precauție, etc. Fiecărei probleme detectate i se asociază o severitate (Low, Medium, High) și un nivel de încredere al detecției (Confidence: Low, Medium, High), ajutând dezvoltatorul să prioritizeze rezultatele.

Instalare și rulare: Bandit se instalează ușor folosind pip:

1. Instalați Bandit cu pip: `pip install bandit`. (Este un pachet Python pur, deci nu necesită altceva. Asigurați-vă totuși că aveți Python 3.x instalat.)

2. Navigați în directorul proiectului Python pe care doriți să îl analizați și rulați Bandit specificând fie un fișier, fie recursiv un folder. Cel mai comun este: `bandit -r .` (unde `-r` indică scanare recursivă, iar `.` este directorul curent). Bandit va analiza toate fișierele `.py` din proiect.
3. Așteptați finalul scanării – pentru proiecte mici durează câteva secunde. Rezultatele sunt afișate direct în terminal.

Interpretarea rezultatelor Bandit: Implicit, Bandit va produce un raport text ordonat pe fișiere și linii cu probleme:

- Fiecare potențială problemă are un cod de identificare (e.g. *B301*), un nivel de *Severity* și de *Confidence*. De exemplu: `B308: import of subprocess with shell=True` ar putea fi Medium Severity, High Confidence (indicând aproape sigur o problemă de securitate moderată).
- Se indică fișierul și numărul liniei unde a fost găsită problema, urmate de descriere. De exemplu, output-ul poate fi:

  ```
  pgsql
  >> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess
  call with shell=True identified, which is a security hazard.
     Severity: Medium   Confidence: High
     File: ./app/util.py   Line: 58
  ```

  Acest mesaj semnalează că la *linia 58 din app/util.py*, scriptul folosește `subprocess.call(..., shell=True)`, ceea ce Bandit consideră o practică riscantă (permite injection de comenzi). Bandit explică pe scurt problema și indică un ID (B602) ce corespunde unei reguli cunoscute (a se vedea documentația Bandit pentru detalii despre fiecare test).

- La final, Bandit oferă un rezumat: număr total de probleme găsite, pe categorii de gravitate.

Informațiile de severitate și încredere ajută la filtrare: de exemplu se pot rula din start doar probleme High (`--severity-level High`) sau se pot exclude probleme cu confidence Low pentru a elimina potențiale fals pozitive.

hardcoded_sql_expressions: Possible SQL injection vector through string-based query construction.
Test ID: B608
Severity: MEDIUM
Confidence: LOW
CWE: CWE-89
File: ./app/api/contexts/application/gateway/repository.py
Line number: 56
More info: https://bandit.readthedocs.io/en/1.7.7/plugins/b608_hardcoded_sql_expressions.html

```
55                          col(app.name).ilike(name),
56                          text(f"""(
57                               json_typeof(app_alias.feature_highlight) = 'array'
58                               AND
59                               EXISTS (
60                                  SELECT 1
61                                  FROM json_array_elements(app_alias.feature_highlight) AS elem
62                                  WHERE elem->>'body' LIKE '{name}'
63                               ))
64                               """)))
65              category = kwargs.pop('category', None)
```

Exemplu: Output Bandit indicând o potențială vulnerabilitate de injecție SQL (regula „hardcoded_sql_expressions" – query SQL construit prin concatenare nesigură). Bandit atribuie fiecărei probleme un cod (B608), severitate (MEDIUM), încredere (LOW) și identificator CWE relevant.

În cadrul laboratorului, rularea Bandit pe aplicații Python va evidenția rapid lucruri precum: zone unde nu se validează inputul, utilizarea modulului `pickle` (deserializare nesigură), sau chiar practici neoptime (ex: criptografie slabă). Studenții pot compara aceste rezultate cu ceea ce ar fi găsit manual. De asemenea, pot ajusta configurația Bandit (prin fișier *bandit.yml* sau opțiuni în linia de comandă) pentru a **exclude false positives** cunoscute sau pentru a include plugin-uri suplimentare.

Flawfinder (C/C++)

Prezentare: *Flawfinder* este un instrument simplu creat de David A. Wheeler pentru scanarea codului sursă C/C++ în vederea identificării **posibilelor vulnerabilități**. Practic, Flawfinder caută în cod *indicatori* de funcții sau constructe riscante și raportează așa-numitele *"hits"* – fragmente de cod potențial vulnerabile – împreună cu o estimare a riscului. Are o listă internă de peste 160 de funcții API C/C++ cunoscute ca problematice (ex: `strcpy`, `sprintf`, `gets`, funcții de memorie, funcții random nesigure, etc.) și folosește și identificatori CWE. Fiecărui *hit* îi este atribuit un nivel de risc de la 0 la 5 (5 fiind maxim risc) în funcție de cât de gravă e utilizarea acelui element în context.

Instalare: Flawfinder fiind un utilitar Python, se poate instala ușor: fie via `pip install flawfinder`, fie pe distribuții Linux direct din managerul de pachete (ex: `sudo apt-get install flawfinder`). După instalare, comanda `flawfinder` devine disponibilă.

Utilizare: Pentru a scana un proiect C/C++, se rulează:

```
bash
flawfinder /cale/catre/directorul_cu_cod
```

sau se poate specifica direct un fișier sursă. Flawfinder va scana recursiv fișierele `.c/.cpp/.h` și va genera un raport text în stdout (sau în fișier, dacă se redirecționează). Se pot adăuga opțiuni:

`-Q` pentru mod silențios (fără banner), `-R` pentru a ordona rezultatele după riscuri, `--html` pentru raport HTML etc.

Interpretarea rezultatelor Flawfinder: Output-ul text al Flawfinder listează fiecare *hit* găsit, cu informații:

- Format general: `path/filename:line [risk] (category) name: message`.
- De exemplu, rulând pe un cod de test, am putea primi:

  ```
  yaml
  test.c:32: [5] (buffer) gets: Does not check for buffer overflows. Use
  fgets() instead.
  test.c:56: [5] (buffer) strcpy: Does not check for buffer overflows.
  test.c:120: [2] (format) printf: If format strings can be influenced by
  an attacker, they can be exploited.
  ...
  ```

  În acest exemplu, la linia 32 din `test.c` s-a găsit un apel `gets` – Flawfinder îl marchează cu risc [5] (maxim) în categoria "buffer" și explică: "*Does not check for buffer overflows. Use fgets() instead.*". Deja, chiar și fără cunoștințe anterioare, mesajul indică clar problema și recomandarea. Similar, la linia 56 `strcpy` este considerat risc 5 (întrucât poate produce overflow dacă nu e folosit cu atenție). La linia 120, `printf` are risc 2 în categoria "format", cu avertizarea standard despre format string vulnerability (dacă formatul provine din exterior).

- După lista de hit-uri, Flawfinder afișează un **rezumat**: număr de *hits* găsite la fiecare nivel de risc (0-5) și numărul total de linii de cod analizate. De exemplu: `[0] 0 hits,`

`[1] 3 hits, [2] 5 hits, [3] 0 hits, [4] 0 hits, [5] 2 hits` arată distribuția pe niveluri.

- Mesajele includ adesea referințe la identificatori **CWE**. De exemplu, în mesajul pentru `gets` ar putea apărea `(CWE-120)` indicând "Buffer Copy without Checking Size of Input (Classic Buffer Overflow)" – ceea ce ajută la documentare ulterioară.

Interpretarea se face similar ca la Bandit: se verifică fiecare hit și se stabilește dacă e un *vulnerabilitate reală* în context. Flawfinder, neavând analiza de flux complexă, poate da unele false positives (de ex. va semnala *strcpy* chiar dacă poate programatorul a verificat manual dimensiunile înainte). Totuși, evidențiază rapid locurile sensibile din cod care merită revizuite manual. În laborator, folosind Flawfinder pe cod C/C++ vom putea vedea imediat *"locurile periculoase"* – este un mod bun de a verifica dacă echipa de programare a evitat funcțiile cunoscute ca riscante. Remedierea problemelor găsite de Flawfinder implică de obicei înlocuirea funcțiilor cu alternative safe (ex: `gets` -> `fgets`, `strcpy` -> `strncpy` cu calculul lungimii corecte, folosirea `snprintf` în loc de `sprintf`, etc.), adică aplicarea bunelor practici de programare sigură.

CodeQL

Prezentare: *CodeQL* este un instrument puternic de analiză statică dezvoltat de GitHub, bazat pe ideea de a trata codul ca pe o bază de date ce poate fi interogată cu un limbaj de query specific (derivat din SQL). În esență, CodeQL este un *motor semantic de analiză a codului* ce permite scrierea de interogări pentru a găsi tipare de vulnerabilitate în cod. GitHub utilizează CodeQL pentru funcționalitatea sa de **code scanning** (securitate avansată în repository-urile GitHub). Avantajul major al CodeQL este flexibilitatea: pe lângă setul de query-uri de securitate predefinite (care acoperă o gamă largă de vulnerabilități cunoscute, de la injecții, buffer overflows, până la erori de logică), utilizatorul poate scrie noi query-uri pentru a detecta vulnerabilități specifice contextului. CodeQL suportă multe limbaje (C/C++, C#, Go, Java, JavaScript/TypeScript, Python, Ruby, Kotlin, etc.).

Instalare: CodeQL poate fi folosit în mai multe moduri:

- **Local CLI:** Descărcați **CodeQL CLI** (disponibil gratuit pentru proiecte open-source). Este un binar Java ce poate fi rulat local. Necesită Java și un compilator pentru limbajul țintă (pentru a construi o bază de date a codului).
- **IDE (VS Code) Extension:** Există o extensie *CodeQL* pentru Visual Studio Code, care oferă un mediu mai comod de rulare a interogărilor și vizualizare a rezultatelor.

- **GitHub Actions:** În proiectele de pe GitHub, cea mai întâlnită utilizare este prin configurarea unui workflow de CodeQL action, care la fiecare push compilează codul, rulează query-urile CodeQL și publică alertele în interfța repository-ului (tab-ul "Security > Code scanning alerts").

În cadrul laboratorului vom descrie procesul folosind CLI local, pentru a vedea pașii implicați:
Utilizare (scanare cu CodeQL CLI):

1. **Crearea bazei de date CodeQL:** Înainte de analiză, CodeQL necesită să construiți o bază de date ce reprezintă codul. Navigați la proiectul de analizat și rulați:

   ```
   bash
   codeql database create db-proiect --language=cpp --source-root=./
   ```

   (Înlocuiți `cpp` cu limbajul relevant, ex: `--language=java` pentru Java, sau specificați multiple limbaje dacă e poliglot.) Comanda de mai sus va compila proiectul (dacă e necesar) și va genera o bază de date CodeQL în folderul `db-proiect`. Practic, CodeQL monitorizează procesul de build al proiectului pentru a popula această bază de date cu informații semantice.

2. **Rularea interogărilor de securitate:** CodeQL vine cu pachete de query-uri gata făcute. GitHub oferă un *repo de query-uri* oficial (`codeql/<lang>-queries`) care include și categoria **Security and Quality**. De exemplu, pentru C/C++ puteți rula:

   ```
   bash
   codeql database analyze db-proiect codeql/cpp-queries:Security --
   format=sarifv2.1.0 --output=rezultate.sarif
   ```

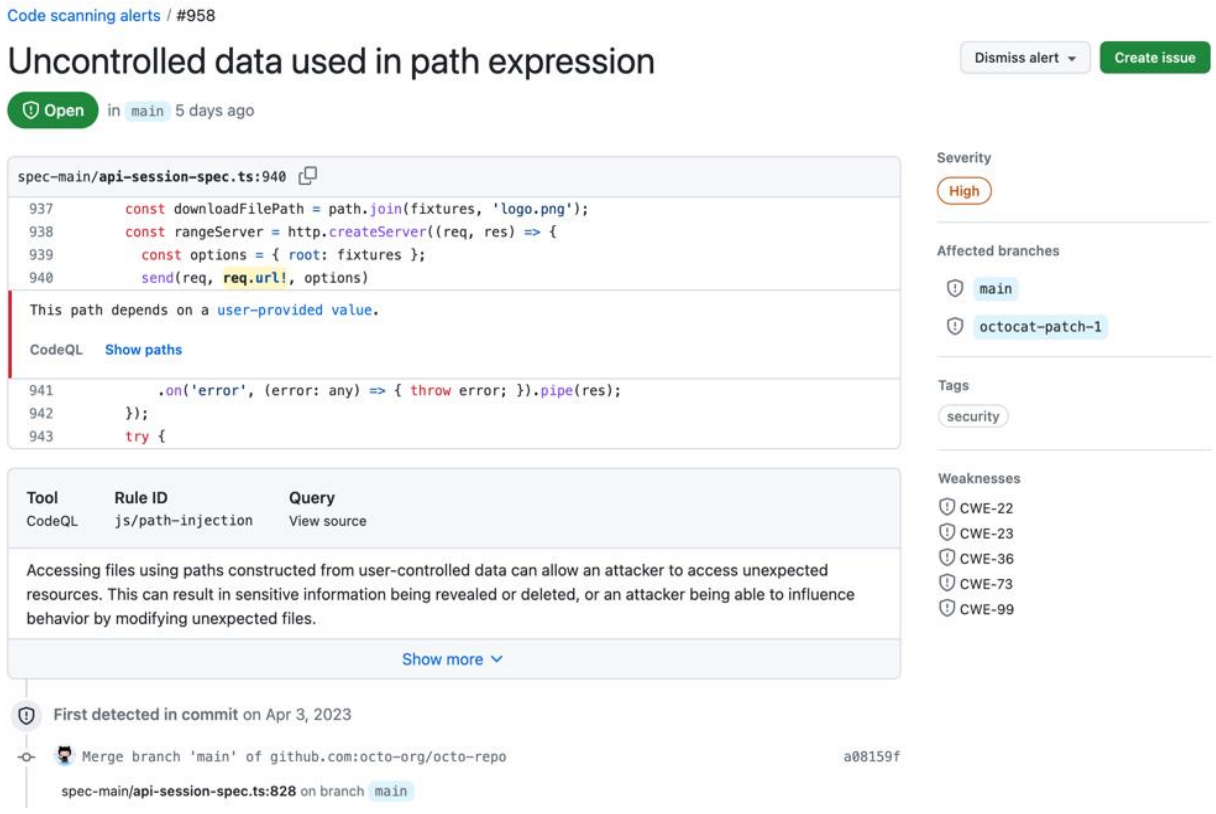
   Aceasta va executa toate interogările din categoria de Securitate pe baza de cod a proiectului și va produce un raport în format SARIF (un format standard pentru rezultate de analiza statică). Formatul SARIF poate fi vizualizat cu diverse unelte (inclusiv importat în GitHub). Alternativ, puteți folosi `--format=CSV` sau `--format=txt` pentru a vedea rapid rezultatele.

3. **Analiza rezultatelor:** Rezultatele conțin lista de alerte de securitate găsite, similar ca structură cu cele din alte unelte (fiecare cu un identificator, o descriere, severitate). Ce este special la CodeQL este că pentru vulnerabilitățile de tip *taint-flow* (injection, XSS, etc.) oferă traseul de execuție: adică indică *sursa* datelor nesigure, pașii prin care trec

(propagare) și *destinația periculoasă* unde ajung. Acest *trace* ajută enorm la înțelegerea vulnerabilității.

4. (Opțional) **Interogări personalizate:** Dacă doriți, puteți scrie propriile query-uri `.ql` pentru a verifica scenarii specifice. De exemplu, ați putea scrie o interogare care să găsească toate locurile unde se deschide un fișier în modul scriere fără a verifica dacă fișierul există (potențial race condition). Acesta este însă un aspect mai avansat și nu este necesar pentru primele laboratoare – ne vom concentra pe rularea interogărilor existente.

Interpretarea și exemple: Multe din alertele CodeQL corespund unor vulnerabilități clasice. De exemplu, CodeQL pentru JavaScript are o regulă care detectează **Path Traversal** – situația în care input-ul utilizatorului este folosit într-o cale de fișier fără validare, permițând atacatorului să iasă din directorul permis. Alertelor li se dă un titlu clar, de ex: "*Uncontrolled data used in path expression*". Vor fi evidențiate fragmentele de cod: de la sursa input-ului (ex: `req.url` în cadrul unei cereri web) până la locul unde este folosit (ex: parametru la `sendFile` fără nici o normalizare). În GitHub, o astfel de alertă arată ca în figura de mai jos.



Exemplu: Alertă de securitate generată de CodeQL (integrat în GitHub) – "Uncontrolled data used in path expression". Se observă codul sursă cu linia problematică evidențiată (linia 940,

unde se folosește `req.url` *direct), severitatea (High) și identificatori de slăbiciuni (CWE-22, CWE-23 – Path Traversal) în partea dreaptă.*

În interpretare, trebuie examinat traseul datelor: de unde provin datele de la linia respectivă și cum ajung acolo. CodeQL furnizează butonul **"Show paths"** (în GitHub UI) care, odată apăsat, afișează secvențele de cod intermediare. Acest tip de analiză de flux evidențiază lucruri pe care uneori un tool simplu ca Flawfinder nu le poate vedea (de exemplu, că variabila a fost sanitizată sau nu într-o altă funcție).

Pentru laborator, utilizarea CodeQL local poate fi mai complexă, așa că este posibil să demonstrăm rezultatele folosind un proiect open-source scanat deja pe GitHub (de exemplu, să arătăm direct în interfata GitHub câteva alerte de securitate și cum se navighează în ele). Scopul este ca studenții să înțeleagă potențialul acestor instrumente avansate: CodeQL nu doar găsește *unde* este problema, dar oferă contextul complet (proveniența datelor, fluxul programului) pentru a înțelege *de ce* este o problemă și cum s-o rezolve.

Studiu de caz și exerciții practice

În această secțiune vom aplica cunoștințele pe un proiect concret și vom compara rezultatele analizei statice manuale vs. automatizate, urmate de procesul de remediere a vulnerabilităților identificate.

Scopul exercițiului: Să se ia un proiect software existent (sau un subset de cod suficient de mare) și să se efectueze atât analiză manuală cât și cu unelte automate, pentru a vedea în practică avantajele fiecărei abordări și pentru a exersa procesul de **identificare** și **remediere** a problemelor de securitate.

Pași propuși pentru activitatea de laborator:

1. **Selectarea aplicației țintă:** Instructorul poate furniza un proiect vulnerabil deliberat (de ex. o aplicație web simplă cu mai multe bug-uri de securitate introduse intenționat – similar cu DVWA, dar poate la scară mică, sau un modul din OWASP Juice Shop, etc.). Alternativ, se poate folosi un proiect open-source real, dar atunci e preferabil unul cunoscut pentru vulnerabilități (o versiune mai veche). Important e ca toți studenții să aibă același cod de analizat pentru a putea discuta ulterior. Se pune la dispoziție codul sursă și o scurtă descriere a funcționalităților, astfel încât participanții să îl poată înțelege.

2. **Analiză statică manuală a codului:** Fiecare echipă va parcurge codul și va nota potențialele vulnerabilități găsite manual, folosind cunoștințele din secțiunile precedente. De exemplu, dacă proiectul este o aplicație web, studenții vor căuta dacă se face validare pe input, dacă se construiesc interogări SQL direct cu parametri, dacă parolele sunt stocate necriptat, dacă se deschid fișiere pe disc folosind căi ce pot fi manipulate, etc. Este util ca echipele să realizeze o listă tabelară cu: *Loc în cod – Tip vulnerabilitate suspectată – Descriere/impact*.
    - Exemplu de identificare manuală: "În `UserController.java`, metoda `login()` ia parametrii `user` și `pass` din solicitare și construiește o interogare SQL concatenând stringuri – posibil **SQL Injection**."
    - Alt exemplu: "În `upload.php`, numele fișierului încărcat de utilizator este folosit direct pentru a salva fișierul pe server, fără a verifica sau normaliza calea – posibil **Path Traversal** (un utilizator ar putea urca un fișier cu numele `../../shell.php`)."
    - Echipele ar trebui să identifice cel puțin 3-5 probleme (numărul depinde de dimensiunea codului și de cât de vulnerabil este intenționat).
3. **Analiza statică automatizată a codului:** După ce au epuizat rezonabil analiza manuală, studenții vor rula instrumentele prezentate anterior pe același cod. Ideal se folosește câte un instrument potrivit pentru tehnologia proiectului:
    - Dacă proiectul e în C/C++ – rulăm Flawfinder (și eventual CodeQL pentru C/C++ dacă e configurat).
    - Dacă e în Python – rulăm Bandit.
    - Dacă e o aplicație web cu mai multe componente (ex: backend Java, frontend JS) – putem rula SonarQube (dacă e instalat în lab) pentru o vedere de ansamblu, plus poate Dependency-Check pentru a vedea vulnerabilități în librării.
    - În cazul în care configurarea SonarQube/CodeQL e dificilă live, instructorul poate furniza un raport generat în prealabil.
    - Studenții trebuie să adune rapoartele generate de unelte și să le compare cu lista lor manuală.
4. **Compararea rezultatelor manual vs. automat:** Fiecare echipă va analiza în ce măsură *instrumentele automate au găsit vulnerabilitățile pe care ei le-au identificat manual* și dacă au raportat și altele în plus. Se vor discuta aspecte precum:
    - Au existat **vulnerabilități pe care tool-urile le-au semnalat, dar pe care echipa nu le-a observat manual**? (De exemplu, poate un student a ratat un `eval()` periculos în cod, dar Bandit l-a raportat.)

- Invers, au fost **probleme identificate manual care nu apar în raportul automat**? (Acest caz e foarte interesant – de exemplu, o logică de autentificare implementată greșit ar putea să nu fie prinsă de nicio regulă automată, dar un ochi uman își dă seama că e o breșă; astfel se evidențiază importanța analizei manuale complementare.)
- Câte dintre alertele raportate automat sunt **false positive**? Studenții ar trebui să examineze fiecare alertă din unelte: este într-adevăr o problemă reală sau codul respectiv e sigur în context? De exemplu, SonarQube ar putea marca o posibilă injectare SQL, dar dacă codul folosește oricum un ORM care scăpa parametrii, alerta ar fi un false positive. Aceste situații trebuie notate.
- Se poate completa un mic tabel comparativ: *Vulnerabilitate X – Găsită manual? (Y/N) – Găsită automat? (Y/N, cu ce unealtă) – Observații*. Acest tabel va evidenția acoperirea combinată a metodelor.

5. **Remedierea vulnerabilităților identificate:** Odată listate toate problemele (confirmate) din cod, următorul pas al exercițiului este **să le remedieze**. Fiecare echipă poate alege una sau două vulnerabilități din cele găsite și să modifice codul pentru a le rezolva, urmând practicile de secure coding:
   - Pentru o vulnerabilitate SQLi – se va modifica codul pentru a utiliza *prepared statements* sau proceduri stocate, eliminând concatenarea directă a input-ului.
   - Pentru XSS – se va aplica filtrare sau escape pe output (ex. folosirea unei funcții de sanitizare HTML).
   - Pentru buffer overflow – se va înlocui funcția nesigură, se va adăuga verificare de limite sau se va schimba arhitectura (poate folosind un container de string cu dimensionare dinamică).
   - Dacă s-au găsit parole hardcodate sau chei API – se vor muta în variabile de mediu sau fișiere de config securizate, etc.
   - În cazul dependențelor vulnerabile (raport Dependency-Check) – se vor actualiza acele biblioteci la versiuni mai noi (dacă timpul permite și este fezabil în proiect).
   - Fiecare remediere se testează rapid (dacă aplicația e rulabilă) ca să nu introducă bug-uri funcționale.

   După modificări, *rulează din nou instrumentele de analiză statică* pe codul actualizat. Ideal, rapoartele ar trebui să nu mai arate vulnerabilitatea respectivă. (De exemplu, dacă înainte Bandit raporta `B608 hardcoded_sql_expression` la linia 56, după fixarea codului acea alertă ar trebui să dispară.) Astfel, studenții primesc confirmarea

automatizată că remedierea a avut efect. Totodată, este o bună practică de workflow DevSecOps: **analiză – remediere – re-scanare** pentru verificare.

6. **Discuție și concluzii pe baza studiului de caz:** Se încurajează echipele să împărtășească ce au învățat:
    - Care vulnerabilitate a fost cea mai surprinzătoare sau greu de depistat manual?
    - Cât de multe probleme a găsit automatizarea față de ochiul uman? Au existat diferențe notabile?
    - Dacă au apărut false positives, cum le-au recunoscut și cum le-ar marca (ex: în SonarQube poți marca o problemă ca *"False Positive"* sau *"Won't Fix"* pentru a nu deranja scorul).
    - Ce soluții de fix au aplicat și dacă a fost ușor/dificil? Au trebuit schimbări majore de cod sau doar mici corecții?
    - În final, studenții ar trebui să aibă o imagine mai clară asupra modului în care analiza statică se încadrează în procesul de dezvoltare: este mult mai eficient să găsești aceste probleme în faza de cod decât să le lași să ajungă exploatabile în producție.

(Notă: Acest studiu de caz poate fi adaptat în funcție de timpul de laborator. Pentru un curs practic, se pot întinde aceste activități pe mai multe sesiuni: una pentru scanare manuală, una pentru scanare automată, una pentru remediere. Important este ca studenții să parcurgă integral ciclul și să vadă beneficiile fiecărui pas.)

Concluzii și bune practici

Analiza statică a codului sursă, atât manuală cât și automatizată, s-a dovedit un instrument esențial în arsenalul dezvoltării de software securizat. În încheiere, sintetizăm câteva concluzii și recomandări:

- **Limitările analizei statice:** Deși foarte valoroasă, analiza statică nu este un panaceu. Unele tipuri de vulnerabilități nu pot fi depistate fără a rula efectiv aplicația (de exemplu, probleme ce depind de configurații de runtime, de interacțiuni complexe sau de starea sistemului – acestea țin de analiza dinamică). Analiza statică operează cu informația disponibilă în cod, deci *nu poate explora toate căile de execuție posibile* la runtime, putând rata bug-uri care apar doar în scenarii foarte particulare. Totodată, instrumentele automate produc uneori *false positive* – alarme pentru fragmente de cod care în realitate nu sunt exploatabile. Aceste rezultate trebuiesc filtrate și validate manual (de aceea

intervenția umană rămâne importantă). În studiul de caz am observat și *false negative*-uri (cazuri în care vulnerabilitatea a fost omisă de unealtă dar prinsă de om, sau vice-versa); de aceea se recomandă întotdeauna o abordare combinată: unelte + review uman

- **Integratea într-un proces DevSecOps:** O bună practică este să integrați analiza statică automată în procesul de build continuu (CI/CD). De exemplu, configurați pipeline-ul de CI astfel încât la fiecare push sau înainte de un merge important, codul să fie scanat cu instrumente precum SonarQube, CodeQL sau altele, iar dacă sunt găsite vulnerabilități severe, build-ul să fie marcat eșuat (quality gate failure). Astfel, problemele sunt semnalate *în timp real* dezvoltatorilor, înainte de a ajunge în ramura principală. Acest lucru creează o cultură în care **securitatea este verificată continuu**, nu lăsată doar pentru etapele finale.
- **Actualizarea dependențelor și Software composition analysis (SCA) continuu:** Asigurați-vă că folosiți frecvent unelte de tip Dependency-Check sau alternative (Snyk, Dependabot, etc.) pentru a monitoriza bibliotecile terțe din proiect. Vulnerabilități noi apar constant în pachetele populare, iar menținerea lor la zi reduce foarte mult suprafața de atac. Includerea unui raport SCA în livrabilele de build (sau folosirea serviciilor care trimit alerte când apare o CVE ce vă afectează) este o bună practică indispensabilă.
- **Cod securizat prin design și revizuiri:** Încurajați aplicarea principiilor de **secure coding** încă din faza de dezvoltare. Urmați ghiduri precum OWASP Secure Coding Guidelines sau CERT Secure Coding Standards pentru limbajul folosit. De exemplu, pentru C++ urmați recomandările CERT (EXPxx-C, MSCxx-C etc.), pentru Java folosți validați inputurile, evitați serializabile nesigure, etc. Dacă programatorii cunosc aceste practici, codul rezultat va avea mai puține probleme de la început. Complementar, instituiți **code review** regulat: orice feature nou ar trebui revizuit de un coleg, având și o checklist de securitate în minte. Code review-urile umane pot prinde aspecte de design (ex: "acest modul nu autentifică cererea înainte să proceseze datele – trebuie adăugată verificare") pe care niciun scanner nu le poate înțelege din context.
- **Educație și cultură de securitate:** Unul din cele mai valoroase "instrumente" este, de fapt, educarea echipei. Asigurați-vă că toți membrii echipei de dezvoltare au cunoștințe de bază despre vulnerabilități și impactul lor. Organizarea unor training-uri OWASP Top 10, exerciții de tip capture-the-flag axate pe securitate, sau rotația membrilor prin rolul de "security champion" al echipei, pot ajuta la crearea unei culturi proactive. Când dezvoltatorii știu *ce* să evite, vor scrie direct cod mai sigur, reducând volumul de probleme raportate de unelte.

- **Politici de cod și quality gates:** Stabiliți metrici clare de calitate a codului ce includ securitatea. De exemplu, puteți impune ca "zero Vulnerabilities Blocker/Critical" înainte de release – adică toate vulnerabilitățile grave semnalate de scanner trebuie rezolvate. SonarQube și altele permit definirea acestor *Quality Gates*. Atenție totuși să calibrați cerințele pentru a fi realiste (să nu blocați pipeline-ul pentru chestii minore, altfel echipa poate fi tentată să ignore în totalitate alertele).
- **Combinarea mai multor unelte:** Fiecare instrument de analiză statică are puncte forte și limitări. De exemplu, un lintern de securitate specific limbajului (Bandit pentru Python) va cunoaște idiom-urile acelui ecosistem mai bine decât un tool generalist. În schimb, un framework ca CodeQL poate descoperi vulnerabilități logice mai complexe. De aceea, în proiectele critice se folosesc adesea *mai multe unelte în paralel*. Nu este redundant – dacă ai două scanere SAST, șansa ca ambele să rateze aceeași problemă scade, și poți corela rezultatele lor pentru o imagine mai completă. Desigur, trebuie gestionat și volumul de rezultate (consolidarea alertelor, eliminarea duplicatelor).
- **Gestionarea vulnerabilităților identificate:** Odată ce analiza statică (sau orice test de securitate) a scos la iveală probleme, este crucial să existe un proces de *management al vulnerabilităților*: triați și prioritizați problemele (ex: criticele se rezolvă imediat, cele medii pot fi planificate în sprintul următor, etc.), deschideți tichete de bug în sistemul de tracking intern pentru fiecare vulnerabilitate de remediat, urmăriți rezolvarea și retestați după fix. În medii agile, vulnerabilitățile ar trebui tratate ca debt tehnic foarte important de plătit rapid.

În final, putem spune că **o abordare eficientă a securității software implică "straturi" multiple**: analiza statică (manuală și automatizată) pentru a curăța codul sursă, analiza dinamică (teste de penetrare, fuzzing) pentru a verifica aplicația în execuție, practici de DevSecOps pentru a integra aceste verificări continuu, și o cultură de dezvoltare care acordă prioritate securității. Adoptând aceste bune practici, riscul vulnerabilităților software scade semnificativ, contribuind la producerea unor aplicații mai sigure și mai fiabile.