VIŠJA STROKOVNA ŠOLA ACADEMIA

MARIBOR

PRIMERJAVA DVEH ORODIJ ZA STATIČNO ANALIZO KODE (SAST) ZA ZAČETNIKE: SONARQUBE CE PROTI SEMGREP

Kandidat: Tim Starčič

Vrsta študija: študent izrednega študija

Študijski program: Informatika

Mentor predavatelj: mag. Ervin Schaff

Mentor v podjetju: mag. Dušan Brglez

Lektor: Ana Novak, prof. slov. in angl.

Maribor, 2026

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisani Tim Starčič sem avtor diplomskega dela z naslovom Primerjava dveh orodij za statično analizo kode (SAST) za začetnike: SonarQube CE proti Semgrep, ki sem ga napisal pod mentorstvom mag. Ervina Schaffa in mag. Dušana Brgleza.

S svojim podpisom zagotavljam, da:

- je predloženo delo izključno rezultat mojega dela,
- sem poskrbel, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia Maribor,
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli, kot moje lastne kaznivo po *Zakonu* o avtorski in sorodnih pravicah (Uradni list RS, št. 16/07 – uradno prečiščeno besedilo, 68/08, 110/13, 56/15 in 63/16 – ZKUASP); prekršek pa podleže tudi ukrepom Višje strokovne šole Academia Maribor skladno z njenimi pravili,
- skladno z 32.a členom ZASP dovoljujem Višji strokovni šoli Academia Maribor objavo diplomskega dela na spletnem portalu šole.

Ptuj, maj 2026 Podpis študenta:

ZAHVALA

Iskreno se zahvaljujem svoji družini za vso podporo, razumevanje in spodbudo skozi celoten čas študija. Njihova potrpežljivost, zaupanje in vera v moje sposobnosti so mi omogočili, da sem vztrajal tudi v zahtevnejših obdobjih ter uspešno zaključil pomembno življenjsko poglavje.

Posebna zahvala gre moji partnerki Lani, ki mi je ves čas stala ob strani, me motivirala ter mi nudila neprecenljivo moralno oporo.

Zahvaljujem se tudi mentorjema, mag. Ervinu Schaffu in mag. Dušanu Brglezu, za vse znanje, ki sem ga pridobil pri njunih urah pouka, ter za mentorstvo pri pripravi diplomskega dela.

Ob tej priložnosti se zahvaljujem tudi šoli Academia Maribor za pridobljeno znanje, strokovno okolje ter podporo pri razvoju mojih kompetenc na področju informatike.

POVZETEK

V diplomskem delu je obravnavano področje statične analize izvorne kode (SAST) kot pomembnega mehanizma za zgodnje odkrivanje varnostnih ranljivosti v programski opremi. Namen raziskave je primerjati delovanje dveh odprtokodnih orodij za statično analizo, SonarQube CE in Semgrep CE, ter ovrednotiti njuno učinkovitost pri zaznavanju tipičnih varnostnih ranljivosti v aplikacijah Java. V teoretičnem delu so raziskani splošni pomen kibernetske varnosti, statične analize kode ter opis in dokumentacija obeh orodij za analizo kode.

V praktičnem delu sta orodji Semgrep CE in SonarQube CE testirani v programskem jeziku Java ter v aplikacijo namerno implementirane ranljivosti, kot so SQL injekcija, XSS (Cross-Site Scripting) in ranljivost tipa Path Traversal. Za prvi del primerjave orodij je bil uporabljen standardizirani testni sklop OWASP Benchmark Java, saj tak pristop omogoča bolj strukturirano in primerljivo vrednotenje rezultatov na enakem naboru testnih primerov. Kot drugi del primerjave je bila ustvarjena lastna aplikacija Java, v kateri sta bila testirana vpliv in funkcionalnost orodij med razvojnim procesom. Takšna razdelitev praktičnega dela je omogočila, da sta bili orodji preverjeni tako na bolj standardiziranem testnem naboru kot tudi na manjšem lastnem razvojnem primeru. S tem je bila lahko primerjava izvedena z vidika uspešnosti zaznavanja ter z vidika dejanske uporabe orodij pri delu z izvorno kodo.

Orodji sta bili pripravljeni v enotnem testnem okolju, nato pa sta bili izvedeni analiza ter sistematična primerjava zaznanih rezultatov. Posebna pozornost je bila namenjena natančnosti zaznavanja, številu lažno pozitivnih rezultatov ter uporabniški izkušnji pri uporabi orodij, predvsem z vidika začetnika.

Cilj diplomskega dela je bil oceniti primernost obeh pristopov za uporabo v izobraževalnem okolju ter pri razvijalcih z manj izkušnjami na področju kibernetske varnosti. Diplomsko delo tako prispeva k razumevanju praktične uporabnosti sodobnih orodij SAST ter njihovih omejitev pri zaznavanju varnostnih ranljivosti.

Ključne besede: *statična analiza kode, SAST, kibernetska varnost, SonarQube CE, Semgrep CE, varnostne ranljivosti, SQL injekcija, XSS, Path Traversal, OWASP Benchmark.*

ABSTRACT

COMPARISON OF TWO STATIC CODE ANALYSIS (SAST) TOOLS FOR BEGINNERS: SONARQUBE CE VS. SEMGREP CE

This diploma thesis addresses the field of static source code analysis (SAST) as an important mechanism for the early detection of security vulnerabilities in software. The aim of the research was to compare the performance of two open-source static analysis tools, SonarQube CE and Semgrep CE, and to evaluate their effectiveness in detecting typical security vulnerabilities in Java applications. The theoretical part examines the general importance of cybersecurity, static code analysis, and the description and documentation of both static analysis tools.

In the practical part, the tools Semgrep CE and SonarQube CE were tested using the Java programming language, and vulnerabilities such as SQL injection, XSS (Cross-Site Scripting), and Path Traversal were intentionally implemented in the application. For the first part of the comparison, the standardised OWASP Benchmark Java test suite was used, as this approach enables a more structured and comparable evaluation of results based on the same set of test cases. In the second part, a custom Java application was developed to test the impact and functionality of the tools during the development process. This division of the practical part made it possible to evaluate the tools both on a more standardised test set and on a smaller custom-developed example. Consequently, the comparison was conducted from the perspective of detection performance as well as from the perspective of the actual use of the tools when working with source code.

Both tools were prepared within a unified testing environment, after which the analysis and systematic comparison of the detected results were carried out. Special attention was given to detection accuracy, the number of false positives, and the user experience when using the tools, particularly from the perspective of a beginner.

The objective of the thesis was to evaluate the suitability of both approaches for use in educational environments and among developers with less experience in the field of cybersecurity. The diploma thesis therefore contributes to an understanding of the practical applicability of modern SAST tools and their limitations in detecting security vulnerabilities.

Keywords: *static code analysis, SAST, cybersecurity, SonarQube CE, Semgrep CE, security vulnerabilities, SQL injection, XSS, Path Traversal, OWASP Benchmark.*

KAZALO VSEBINE

1 UVOD 11

1.1 Opis področja in opredelitev problema 11
1.2 Namen, cilji in osnovne trditve 11
1.3 Predpostavke in omejitve 12
1.4 Uporabljene raziskovalne metode 13

2 KIBERNETSKA VARNOST IN VARNOST APLIKACIJ 14

2.1 Pomen varnosti programske kode 14
2.1.1 Digitalizacija in odvisnost od programske opreme 14
2.1.2 Posledice varnostnih ranljivosti 14
2.1.3 Programska koda kot izvor ranljivosti 15
2.1.4 Potreba po sistematičnem pristopu k varnosti 15
2.2 OWASP in klasifikacija ranljivosti (CWE) 16
2.2.1 OWASP kot referenčni varnostni okvir 16
2.2.2 Common Weakness Enumeration (CWE) 17
2.2.3 Povezava med OWASP Top 10 in CWE 17
2.3 Najpogostejše varnostne ranljivosti 18
2.3.1 SQL Injection 18
2.3.2 Cross-Site Scripting (XSS) 19
2.3.3 Path Traversal 20
2.4 Koncept varnega razvoja 21
2.4.1 Vključevanje varnosti v življenjski cikel razvoja programske opreme (SDLC) 21
2.4.2 Varnostne prakse v fazi implementacije 22
2.4.3 Preverjanje in zagotavljanje varnosti 23

3 STATIČNA ANALIZA KODE (SAST) 25

3.1 Opredelitev statične analize kode 25
3.1.1 Definicija in osnovni namen statične analize (SAST) 25
3.1.2 Pristopi testiranja glede na dostop do kode: white-box, black-box, grey-box 25
3.1.3 Umestitev SAST med druge pristope testiranja (SAST vs. DAST) 26
3.2 Delovanje statične analize 27
3.2.1 Vhod in izhod statične analize 27
3.2.2 Tehnike analize kode 27
3.2.3 Pravila in zaznavanje ranljivosti 28
3.3 Prednosti in omejitve SAST 28
3.3.1 Prednosti SAST 28
3.3.2 Omejitve SAST 29

3.4 METRIKE VREDNOTENJA (PRECISION, RECALL, F1) 29
3.4.1 Osnovni pojmi (TP, FP, TN, FN) *29*
3.4.2 Precision in Recall *30*
3.4.3 F1 ..... *31*
4 PREDSTAVITEV ORODIJ ZA STATIČNO ANALIZO 32
4.1 SONARQUBE 32
4.1.1 Namen orodja in vloga v procesu razvoja programske opreme *32*
4.1.2 Način delovanja SonarQube Community Build pri analizi kode *33*
4.1.3 Pravila in kakovostni profili *35*
4.1.4 Podprti jeziki in pomen izbire jezika na analizo *36*
4.1.5 Varnostne zmožnosti in omejitev obsega predstavitve *36*
4.1.6 Utemeljitev izbire SonarQube za praktični del diplomskega dela *37*
4.2 SEMGREP 38
4.2.1 Namen orodja in vloga v procesu razvoja programske opreme *38*
4.2.2 Način delovanja Semgrep CE pri analizi kode *39*
4.2.3 Podprti jeziki in pomen izbire jezika za analizo *40*
4.2.4 Varnostne zmožnosti in omejitev obsega predstavitve *41*
4.2.5 Utemeljitev izbire Semgrep CE za praktični del diplomskega dela *41*
4.3 PRIMERJALNI TEORETIČNI PREGLED ORODIJ 42
4.3.1 Primerjava načina delovanja orodij *42*
4.3.2 Primerjava pravil, nastavitev in prilagodljivosti analize *42*
4.3.3 Primerjava podprtih jezikov in obsega varnostnih zmožnosti *43*
4.3.4 Teoretična primernost orodij za začetnike *43*
4.3.5 Stroškovna tabela obeh orodij *44*
5 PRIPRAVA TESTNEGA OKOLJA 47
5.1 OPIS TESTNEGA OKOLJA 47
5.1.1 OWASP Benchmark – Java *47*
5.1.2 Lastna Java aplikacija *48*
5.1.3 Razlog za uporabo obeh pristopov *48*
5.2 PRIPRAVA ORODIJ ZA ANALIZO 49
5.2.1 SonarQube CE *49*
5.2.2 Semgrep *55*
5.2.3 Primerjava priprave obeh orodij za analizo *58*
6 IZVEDBA ANALIZE 59
6.1 ANALIZA NA OWASP BENCHMARK 59
6.1.1 Priprava projekta za analizo *59*

6.1.2 Izvedba analize z orodjem SonarQube CE *59*
6.1.3 Izvedba analize z orodjem Semgrep CE *68*
6.1.4 Način beleženja rezultatov za preverjanje H1 in H2 ..... *70*
6.2 ANALIZA LASTNE JAVA APLIKACIJE 77
6.2.1 Zasnova simuliranega razvojnega primera *78*
6.2.2 Izhodiščna različica aplikacije brez uporabe Semgrep CE *79*
6.2.2 Različica aplikacije s sprotno uporabo Semgrep CE *82*
6.2.3 Primerjava rezultatov za preverjanje H4 ..... *84*
7 SKLEP 86
8 LITERATURA IN VIRI 89
9 PRILOGE 93

KAZALO SLIK

Slika 1: Potek analize kode v okolju SonarQube Community Build 35

Slika 2: Predogled Semgrep AppSec platforme Vir: (https://semgrep.dev/docs/introduction) 39

Slika 3: Stran za prenos SonarQube CE Vir: (https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/) 49

Slika 4: Lokalna mapa SonarQube Vir: Lasten vir 50

Slika 5: Mapa bin/windows-x86-64 Vir: Lasten vir 50

Slika 6: Zagon SonarQube v ukazni vrstici po zagonu StartSonar.bat Vir: Lasten vir 51

Slika 7: Prijavna stran SonarQube na naslovu http://localhost:9000 Vir: (http://localhost:9000/sessions/new?return_to=%2F) 52

Slika 8: Ustvarjanje novega projekta v vmesniku Vir: (http://localhost:9000/projects/create?mode=manual) 52

Slika 9: Izbira metode analize projekta Vir: (http://localhost:9000/tutorials?id=example) 53

Slika 10: Generiranje tokena Vir: (http://localhost:9000/tutorials?id=example&selectedTutorial=local) 53

Slika 11: Token Vir: (http://localhost:9000/tutorials?id=example&selectedTutorial=local) 54

Slika 12: Osnovna konfiguracija v datoteki sonar-project.properties v mapi projekta Vir: Lasten vir 54

Slika 13: Ukaza semgrep --version in pip –version nista prepoznana Vir: Lasten vir 55

Slika 14: Začetek namestitve orodja Semgrep Vir: Lasten vir 56

Slika 15: Uspešna namestitev orodja Semgrep Vir: Lasten vir 56

Slika 16: Opozorilo pri preverjanju različice Semgrepa s py -m Vir: Lasten vir 57

Slika 17: Uspešen izpis različice Semgrepa Vir: Lasten vir 57

Slika 18: Primer ukaza za analizo Java projekta s Semgrepom Vir: Lasten vir 57

Slika 19: Ustvarjanje projekta OWASP Benchmark Vir: Lasten vir 60

Slika 20: Pridobitev tokena za projekt Vir: Lasten vir 61

Slika 21: Datoteka sonar-project.properties v korenski mapi projekta Vir: Lasten vir 62

Slika 22: Konfiguracija datoteke sonar-project.properties za naš projekt Vir: Lasten vir 62

Slika 23: Uspešno prevajanje izvorne kode projekta OWASP Benchmark Vir: Lasten vir 63

Slika 24: Uspešen zagon ukaza sonar-scanner Vir: Lasten vir 64

Slika 25: Projekt OWASPBenchmark v SonarQube CE Vir: Lasten vir 64

Slika 26: Osnovni pregled rezultatov analize projekta OWASP Benchmark v SonarQube CE Vir: Lasten vir 65

Slika 27: Razvrstitev zaznanih opozoril v zavihku Issues Vir: Lasten vir 66

Slika 28: Podrobnejši prikaz posameznega opozorila v zavihku Issues Vir: Lasten vir 66

Slika 29: Primer zaznave v zavihku Security Hotspots Vir: Lasten vir 67

Slika 30: Ukaz za zagon analize nad projektom Vir: Lasten vir 68

Slika 31: Rezultat analize projekta z orodjem Semgrep Vir: Lasten vir 69

Slika 32: Primer zaznave SQL Injection v orodju Semgrep Vir: Lasten vir 69
Slika 33: Primer zaznave Path Traversal v orodju Semgrep Vir: Lasten vir....69
Slika 34: Primer zaznave XSS v orodju Semgrep Vir: Lasten vir 70
Slika 35: Rezultat SonarQube CE nad testom BenchmarkTest00018.java Vir: Lasten vir....73
Slika 36: Rezultat Semgrepa nad testom BenchmarkTest00018.java Vir: Lasten vir 74
Slika 37: Struktura lastne Java aplikacije v okolju Visual Studio Code Vir: Lasten vir 79
Slika 38: Rezultati analize izhodiščne različice v Semgrep CE 1/2 Vir: Lasten vir 80
Slika 39: Rezultati analize izhodiščne različice v Semgrep CE 2/2 Vir: Lasten vir 81
Slika 40: Rezultati analize izboljšane različice s Semgrep CE Vir: Lasten vir....83

KAZALO TABEL

Tabela 1: Confusion matrix....30
Tabela 2: Cene produktov SonarQuba 44
Tabela 3: Cene produktov Semgrepa 46
Tabela 4: Rezultati analize vzorčnega sklopa z obema orodjema....71
Tabela 5: Metrike uspešnosti zaznavanja s SonarQube CE 74
Tabela 6: Izračun metrik analize s SonarQube CE 75
Tabela 7: Metrike uspešnosti zaznavanja s Semgrep 76
Tabela 8: Izračun metrik analize s Semgrep CE 76
Tabela 9: Izračun zmanjšanja napak v razvojnem procesu pri sprotni uporabi Semgrep CE 84

1 UVOD

1.1 Opis področja in opredelitev problema

Programska oprema je danes del skoraj vseh digitalnih storitev, ki jih uporabljamo vsak dan. Sem sodijo spletne aplikacije, mobilne rešitve in informacijski sistemi v podjetjih. Ker so programske rešitve vse bolj kompleksne, se povečuje tudi možnost, da se v izvorni kodi pojavijo varnostne napake. Takšne napake lahko povzročijo nepooblaščen dostop do podatkov, razkritje občutljivih informacij ali celo popoln prevzem sistema. Med pogostimi primeri so še vedno SQL injekcija, napadi tipa cross-site scripting (XSS) in Path Traversal, čeprav so te ranljivosti znane že vrsto let.

Zaradi tega postaja varnost programske kode vedno pomembnejši del razvoja programske opreme. Eden izmed načinov za odkrivanje težav še pred zagonom aplikacije je statična analiza izvorne kode oziroma Static Application Security Testing (SAST). Pri takem pristopu se koda pregleda brez izvajanja programa. Orodja pri tem iščejo vzorce, ki bi lahko pomenili varnostno tveganje ali neustrezno programsko prakso. Glavna prednost takšne analize je, da omogoča zgodnje odkrivanje težav, še preden aplikacija pride v produkcijsko okolje.

Danes obstaja več orodij za statično analizo kode. Nekatera so odprtokodna, druga komercialna. Med seboj se razlikujejo po načinu delovanja, zahtevnosti nastavitve, obsegu pravil in prikazu rezultatov. Za začetnike, študente in manj izkušene razvijalce izbira takega orodja ni vedno preprosta. Pogosto se pojavi vprašanje, katero orodje dovolj dobro zaznava ranljivosti, obenem pa ostane razumljivo in preprosto za uporabo.

V tem diplomskem delu se osredotočamo na primerjavo dveh odprtokodnih orodij za statično analizo kode, in sicer SonarQube CE ter Semgrep CE, ki ju bomo v nadaljevanju omenjali kot SonarQube CE in Semgrep CE. Raziskava proučuje, kako učinkovito zaznavata izbrane tipične ranljivosti in kako primerna sta za uporabo v izobraževalnem okolju oziroma za začetnike na področju razvoja programske opreme.

1.2 Namen, cilji in osnovne trditve

Namen diplomskega dela je primerjati dve odprtokodni orodji za statično analizo kode, SonarQube CE in Semgrep CE, ter ugotoviti, kako učinkovito zaznavata tipične varnostne ranljivosti v aplikacijah Java. Delo ni usmerjeno samo v tehnični vidik zaznavanja napak,

ampak tudi v uporabniško izkušnjo. Pri tem je poudarek predvsem na začetnikih in študentih, ki se s področjem kibernetske varnosti šele spoznavajo.

Cilj raziskave je na standardnem testnem sklopu izvesti analizo z obema orodjema. Dobljeni rezultati bodo primerjani glede na število zaznanih ranljivosti, način prikaza rezultatov, možnost prilagajanja pravil in splošno zahtevnost uporabe. Posebej bomo pozorni tudi na lažno pozitivne in lažno negativne rezultate, saj lahko ti precej vplivajo na dejansko uporabnost orodja. Nato bomo izdelali tudi lastno aplikacijo Java, na kateri bomo spremljali, kako lahko vključitev orodij za statično analizo kode vpliva na razvojni tok projekta.

Diplomsko delo izhaja iz naslednjih osnovnih trditev:

1. Med orodjema SonarQube CE in Semgrep CE obstajajo razlike pri zaznavanju določenih vrst ranljivosti.

2. Semgrep zaradi možnosti definiranja lastnih pravil omogoča večjo prilagodljivost analize.

3. SonarQube CE zaradi grafičnega uporabniškega vmesnika in pregledne predstavitve ponuja bolj intuitivno uporabniško izkušnjo za začetnike.

Z raziskavo želimo preveriti, v kolikšni meri te trditve držijo v praksi. Zanima nas tudi, ali je mogoče eno izmed orodij opredeliti kot primernejše za uporabo v izobraževalnem okolju.

Hipoteze, ki smo jih v okviru tega diplomskega dela obravnavali, so naslednje:

H1: Semgrep bo na standardnih testnih primerih dosegel enako ali višjo natančnost (F1) kot SonarQube CE za izbrane CWE (SQLi, XSS, Path Traversal) pri istem programskem jeziku.

H2: SonarQube CE bo generiral več lažno pozitivnih rezultatov kot Semgrep CE na enakem vzorčnem sklopu.

H3: Začetnik bo hitreje namestil in povezal Semgrep (CLI/IDE/CI) kot SonarQube CE ter hitreje razumel razlago napak.

H4: Vključitev izbranega orodja v razvojni tok zmanjša število vnesenih ranljivosti v novi kodi za ≥ 30 % v primerjavi z izhodiščem brez SAST v kratkem simuliranem sprintu.

1.3 Predpostavke in omejitve

Pri raziskavi izhajamo iz predpostavke, da je delovanje orodij za statično analizo kode mogoče primerjati na osnovi analize enake testne aplikacije z namerno vgrajenimi ranljivostmi. Predpostavljamo tudi, da sta orodji pravilno nameščeni in uporabljeni v skladu z njunimi

osnovnimi priporočili ter da izbrane ranljivosti predstavljajo tipične primere varnostnih napak v spletnih aplikacijah.

Raziskava je omejena na odprtokodni različici orodij SonarQube CE in Semgrep CE, zato napredne komercialne funkcionalnosti niso vključene v primerjavo. Analiza je izvedena na eni namensko razviti aplikaciji Java, kar pomeni, da rezultatov ni mogoče neposredno posplošiti na vse vrste projektov ali programske jezike. Dodatno omejitev predstavlja tudi subjektivna ocena uporabniške izkušnje, saj temelji na praktični uporabi orodij v okviru tega diplomskega dela.

Kljub navedenim omejitvam raziskava omogoča realen vpogled v razlike med obema orodjema ter njuno primernost za začetnike.

1.4 Uporabljene raziskovalne metode

Pri izdelavi diplomskega dela je bila uporabljena kombinacija teoretičnega in praktičnega pristopa. V teoretičnem delu je bila uporabljena deskriptivna metoda, s katero smo proučili strokovno literaturo s področja statične analize kode, varnostnih ranljivosti ter kibernetske varnosti. Na osnovi pregleda literature so bili opredeljeni ključni pojmi ter predstavljene osnovne značilnosti orodij SonarQube CE in Semgrep CE.

Osrednji del raziskave temelji na eksperimentalnem pristopu. Razvita je bila testna aplikacija Java z namerno implementiranimi varnostnimi ranljivostmi, kot so SQL injekcija, XSS in Path Traversal. Na tej aplikaciji sta bili izvedeni analizi z obema obravnavanima orodjema. Postopek analize je potekal v enakem testnem okolju, da bi zagotovili primerljivost rezultatov.

Za primerjavo rezultatov je bila uporabljena komparativna metoda. Upoštevane so bile zaznane ranljivosti, način predstavitve rezultatov, možnost prilagajanja pravil ter zahtevnost uporabe orodij. Posebna pozornost je bila namenjena tudi morebitnim lažno pozitivnim ali lažno negativnim zaznavam, saj te pomembno vplivajo na uporabnost orodja v praksi.

Na osnovi zbranih podatkov je bila izvedena analiza in oblikovana končna ocena primernosti obeh orodij za uporabo v izobraževalnem oziroma začetniškem okolju.

2 KIBERNETSKA VARNOST IN VARNOST APLIKACIJ

2.1 Pomen varnosti programske kode

2.1.1 Digitalizacija in odvisnost od programske opreme

V zadnjih desetletjih je programska oprema postala temelj delovanja številnih organizacij, storitev in informacijskih sistemov. Poslovni procesi, finančne transakcije, upravne storitve in komunikacija so danes močno odvisni od digitalnih rešitev. Zaradi tako široke uporabe programske opreme je njena zanesljivost zelo pomembna, saj vpliva na nemoteno delovanje družbe in vsakdanjih storitev (Zgubič, 2015).

Tudi programska oprema, ki je dobro testirana in jo razvijajo izkušeni razvijalci, lahko še vedno vsebuje presenetljivo veliko napak. Nekatere med njimi med običajnim testiranjem sploh niso odkrite. Takšne napake lahko vplivajo na zanesljivost sistema, pogosto pa tudi na njegovo varnost. Prav zato je zgodnje odkrivanje napak pomemben del zagotavljanja kakovosti programske opreme. V sodobnem razvoju imajo zaradi tega vedno večjo vlogo avtomatizirani pristopi za preverjanje kode, med katere spada tudi statična analiza (Hovemeyer in Pugh, 2004; Zgubič, 2015).

2.1.2 Posledice varnostnih ranljivosti

Varnostne ranljivosti v programski opremi lahko povzročijo resne finančne, organizacijske in tudi strateške posledice. Kadar pride do izkoriščanja ranljivosti, to pogosto pomeni neposredne finančne izgube, stroške sanacije in motnje pri poslovanju. Posledice pa niso vedno samo kratkoročne. Pogosto se pokažejo tudi v izgubi ugleda, zmanjšanem zaupanju uporabnikov in višjih stroških upravljanja varnosti v prihodnje (Romanosky, 2016).

Z vidika informacijske varnosti ranljivosti ogrožajo temeljna načela zaupnosti, celovitosti in razpoložljivosti informacijskih sistemov. Prav zato je njihovo obvladovanje pomemben del varnega delovanja vsake organizacije. Standardi in smernice s področja informacijske varnosti poudarjajo, da je sistematičen pristop k prepoznavanju in odpravljanju ranljivosti nujen za zmanjševanje tveganj ter za zagotavljanje stabilnega in varnega informacijskega okolja (NIST, 2020).

Težava postane še večja, če ranljivosti niso odkrite ali odpravljene pravočasno. Daljše kot je obdobje izpostavljenosti, večja je možnost, da bo ranljivost tudi dejansko izkoriščena. To povečuje varnostno tveganje in lahko sproži več povezanih incidentov. Posledice se tako ne kažejo le na ravni posamezne aplikacije, ampak lahko vplivajo tudi na širše digitalno okolje organizacije. S tem rastejo tudi stroški upravljanja varnosti, odzivanja na incidente in splošnega obvladovanja tveganj. Zaradi tega ranljivosti niso problem samo za posamezen sistem, ampak predstavljajo širše tveganje tudi za organizacije in programsko industrijo kot celoto (Algarni, 2022).

2.1.3 Programska koda kot izvor ranljivosti

Varnostne ranljivosti v informacijskih sistemih pogosto izvirajo neposredno iz napak v izvorni kodi. Do njih lahko pride zaradi nepravilne izvedbe posamezne funkcionalnosti, pomanjkljivega preverjanja vhodnih podatkov ali napačne uporabe programskih konstrukcij. Veliko takšnih napak med razvojem ostane neopaženih in se pokaže šele pri dejanski uporabi sistema. Takrat pa lahko že predstavljajo resno varnostno tveganje (Hovemeyer in Pugh, 2004).

K nastanku ranljivosti veliko prispeva tudi kompleksnost sodobnih programskih sistemov. Z večanjem količine kode, številom povezav med posameznimi deli sistema in dolgotrajnim razvojem se povečuje možnost za pojav tehničnih nepravilnosti. Sčasoma lahko pride tudi do upada kakovosti programske rešitve. Takšna odstopanja od dobrih praks slabšajo vzdržljivost in prilagodljivost sistema, obenem pa povečujejo možnost napak, ki se lahko razvijejo v varnostne ranljivosti (Franco, 2020).

Prav zaradi tega standardi informacijske varnosti poudarjajo, da je treba varnostne kontrole vključevati že v fazi razvoja. Sistematično preverjanje kode in sprotno obvladovanje ranljivosti sta pomembna koraka pri gradnji zanesljive in varne programske rešitve (NIST, 2020).

2.1.4 Potreba po sistematičnem pristopu k varnosti

Varnost programske opreme ne bi smela biti nekaj, s čimer se ukvarjamo šele na koncu razvoja. Starejši pristop »penetrate and patch«, pri katerem se pomanjkljivosti odpravlja šele po zaključku razvoja ali po nastanku incidenta, se danes kaže kot manj učinkovit in tudi dražji.

Tak način dela pogosto pomeni ponavljanje enakih ranljivosti in višje stroške vzdrževanja (Humayun et al., 2022).

Razvoj programske opreme je zato varneje načrtovati tako, da se varnostne aktivnosti vključijo že v fazi zahtev, načrtovanja, implementacije, testiranja in poznejšega vzdrževanja. Tako so varnostne zahteve, ocena tveganj in varnostne kontrole prisotne v vsaki fazi razvoja. Prednost tega je predvsem v tem, da se napake odkrijejo prej in da kasnejši popravki niso tako zahtevni (Humayun et al., 2022).

2.2 OWASP in klasifikacija ranljivosti (CWE)

2.2.1 OWASP kot referenčni varnostni okvir

Varnostne ranljivosti spletnih aplikacij se pogosto razvrščajo glede na vrsto napake in njen vpliv na informacijski sistem. Eden izmed najbolj uveljavljenih referenčnih okvirov na tem področju je pripravil OWASP (Open Worldwide Application Security Project). Projekt OWASP Top 10: 2021 vsebuje seznam desetih najresnejših varnostnih tveganj za spletne aplikacije. Temelji na analizi podatkov iz realnega okolja in upošteva dejavnike, kot so pogostost pojavljanja, možnost izkoriščanja in tehnični vpliv na sistem (OWASP, 2021).

V različici iz leta 2021 so ranljivosti razdeljene v deset kategorij. Med njimi so Broken Access Control, Cryptographic Failures, Injection, Insecure Design, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, Security Logging and Monitoring Failures ter Server-Side Request Forgery. Takšna razvrstitev omogoča boljši pregled nad najpogostejšimi in najpomembnejšimi varnostnimi težavami sodobnih spletnih aplikacij (OWASP, 2021).

Za to diplomsko delo je posebej pomembna kategorija A03: Injection. V to skupino sodijo ranljivosti, kot sta SQL Injection, Cross-Site Scripting (XSS) in manipulacija poti do datotek oziroma Path Traversal. Njihova skupna značilnost je, da aplikacija nezaupanja vredne vhodne podatke vključi v interpreter ali sistemski ukaz brez ustrezne validacije, filtriranja ali jasnega ločevanja podatkov od ukazov. Prav zaradi tega so te ranljivosti pogosto med najnevarnejšimi in tudi med najpogosteje zlorabljenimi (OWASP, 2021).

2.2.2 Common Weakness Enumeration (CWE)

Common Weakness Enumeration oziroma CWE je standardizirana referenčna baza slabosti v programski kodi, ki jo razvija in vzdržuje organizacija MITRE. Njen namen je ponuditi enoten način za opis tehničnih slabosti, ki lahko vodijo do varnostnih ranljivosti v informacijskih sistemih. Vsaka slabost v bazi ima svojo identifikacijsko številko, opis in primere. Zaradi tega je mogoče slabosti bolj dosledno razvrščati, primerjati in povezovati med različnimi sistemi ter orodji za analizo kode (MITRE, n. d.).

Pri CWE je pomembna tudi razlika med pojmoma slabost in ranljivost. CWE se osredotoča na slabosti v izvorni kodi ali zasnovi sistema, ki lahko predstavljajo možno pot za napad. Ranljivost pa navadno pomeni konkretno napako v določenem sistemu, ki jo je mogoče tudi dejansko izkoristiti. Takšna ločitev je uporabna zato, ker omogoča natančnejše razumevanje varnostnih problemov. Orodja za analizo kode namreč običajno zaznavajo prav slabosti, te pa je nato mogoče povezati z bolj znanimi razredi napak (MITRE, n. d.).

CWE predstavlja osnovo tudi za številne druge varnostne standarde, poročila in orodja. Uporablja se na primer v povezavi z OWASP Top 10 in tudi v varnostnih modulih orodij za statično analizo kode. Povezovanje slabosti s standardiziranimi identifikatorji pomaga pri poenotenju terminologije in metod dela. Poleg tega izboljša sledljivost varnostnih ukrepov med različnimi fazami razvoja in testiranja programske opreme (MITRE, n. d.).

2.2.3 Povezava med OWASP Top 10 in CWE

OWASP Top 10 in Common Weakness Enumeration (CWE) se razlikujeta predvsem po ravni, na kateri obravnavata varnostne težave. OWASP Top 10 predstavlja širši pregled najpomembnejših varnostnih tveganj v spletnih aplikacijah. Posamezne napake združuje v večje kategorije, ki služijo kot pomoč pri določanju prioritet in usmerjanju varnostnih aktivnosti. Njegov namen je predvsem ozaveščanje ter splošno usmerjanje razvoja varnejših aplikacij (OWASP, 2021).

CWE je po drugi strani bolj tehnično usmerjen. Osredotoča se na konkretne slabosti v kodi ali zasnovi sistema. Vsaka slabost ima svoj identifikator in natančen opis, kar omogoča bolj dosledno tehnično analizo ter primerjavo med različnimi sistemi in orodji. Zaradi tega je CWE

uporaben predvsem takrat, ko želimo napake razumeti bolj podrobno in jih natančno razvrstiti (MITRE, n. d.).

Oba pristopa sta med seboj tesno povezana. Posamezne kategorije v OWASP Top 10 temeljijo na skupinah sorodnih slabosti CWE. Kot smo omenili v poglavju 2.2.1, kategorija A03: Injection na primer vključuje več različnih tehničnih slabosti, med katere sodijo SQL Injection, Cross-Site Scripting in Path Traversal. V okviru CWE so te slabosti opredeljene ločeno, vsaka s svojim identifikatorjem. OWASP torej združuje tehnične slabosti v širše varnostne sklope, medtem ko jih CWE natančneje razčleni (OWASP, 2021; MITRE, n. d.).

Za to diplomsko delo je ta povezava pomembna zato, ker orodja za statično analizo kode zaznavajo konkretne slabosti na ravni CWE. Njihov pomen pa se pogosto razume v okviru širših kategorij OWASP Top 10. Skupaj zato omogočata dobro povezavo med tehničnim odkrivanjem napak in širšim razumevanjem varnostnih tveganj.

2.3 Najpogostejše varnostne ranljivosti

V nadaljevanju so predstavljene izbrane ranljivosti iz kategorije A03: Injection po OWASP Top 10 (2021), ki so vključene tudi v praktični del raziskave. Opisane so njihove značilnosti, mehanizem delovanja ter tipične posledice za informacijski sistem.

2.3.1 SQL Injection

SQL Injection predstavlja eno najpogostejših in najnevarnejših varnostnih ranljivosti spletnih aplikacij. Do ranljivosti pride, kadar aplikacija uporabniški vnos neposredno vključuje v SQL-poizvedbo brez ustrezne validacije ali parametrizacije. V takšnem primeru lahko napadalec z vnosom posebej oblikovanih podatkov spremeni strukturo poizvedbe in s tem vpliva na izvajanje ukaza nad podatkovno bazo. Po opredelitvi OWASP se ranljivosti tipa injection pojavijo, kadar je nezaupanja vreden podatek poslan interpreterju kot del ukaza ali poizvedbe, kar lahko omogoči izvajanje neželenih ukazov ali dostop do podatkov brez ustrezne avtorizacije (OWASP, 2021).

Posledice napada SQL Injection so lahko resne, saj napadalcu omogočajo razkritje občutljivih podatkov, njihovo spremembo ali brisanje. V določenih primerih lahko napad vodi tudi do popolnega prevzema aplikacijskega strežnika. Zaradi velike razširjenosti in pomembnega

vpliva na varnost sistemov OWASP SQL Injection uvršča med najkritičnejše varnostne ranljivosti spletnih aplikacij (OWASP, 2021).

Primer napada:

Ranljivost se pojavi, kadar aplikacija sestavi SQL-poizvedbo tako, da uporabniški vnos neposredno združi z besedilom ukaza, na primer:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'geslo'.
```

Če aplikacija ne uporablja parametriziranih poizvedb, se vrednost, ki jo vnese uporabnik, obravnava kot del SQL-ukaza. Napadalec lahko vnese niz:

```
SELECT * FROM users WHERE username = 'admin' OR '1'= '1'.
```

V tem primeru se izvorna poizvedba logično spremeni v obliko, kjer pogoj '1'='1' vedno drži. Ker je ta izraz vedno resničen, podatkovna baza vrne vse zapise iz tabele uporabnikov ali omogoči prijavo brez preverjanja dejanskega gesla. Ranljivost torej ne nastane zaradi napake podatkovne baze, temveč zaradi nepravilnega sestavljanja poizvedbe v aplikacijski kodi, kjer podatki in ukazi niso ustrezno ločeni (OWASP, 2021).

2.3.2 Cross-Site Scripting (XSS)

Cross-Site Scripting oziroma XSS je varnostna ranljivost, pri kateri aplikacija v spletno stran vključi nepreverjen ali neustrezno obdelan uporabniški vnos. Zaradi tega se lahko v brskalniku drugega uporabnika izvede zlonamerna skripta. Do takšne napake pride, ko aplikacija podatkov pred prikazom v vsebini HTML ne filtrira ali pravilno kodira. OWASP XSS opisuje kot primer, ko aplikacija v spletno stran vključi nezaupanja vredne podatke brez ustrezne validacije ali izhodnega kodiranja, s čimer napadalcu omogoči izvajanje poljubne kode JavaScript v uporabnikovem brskalniku (OWASP, 2021).

Posledice napada XSS so lahko zelo resne. Med pogostimi primeri so kraja sejnih piškotkov, prevzem uporabniškega računa, preusmerjanje na zlonamerne strani in prikaz lažnih prijavnih obrazcev. OWASP razlikuje med tremi osnovnimi oblikami te ranljivosti. To so reflektirani oziroma reflected XSS, shranjeni oziroma stored XSS in DOM-based XSS. Posebej nevaren je shranjeni XSS, ker se zlonamerna koda trajno shrani v aplikaciji in se izvede vsakič, ko uporabnik odpre ranljivo vsebino (OWASP, 2021).

Primer napada:

Ranljivost se pojavi, kadar aplikacija uporabniški vnos neposredno vključi v izhod HTML, na primer v polje za prikaz komentarja. Če uporabnik vnese naslednji niz:

```
<script>alert('Napad')</script>
```

in aplikacija tega vnosa ne kodira ali filtrira, se koda JavaScript izvede v brskalniku vseh uporabnikov, ki si ogledajo stran. Na ta način napadalec pridobi možnost izvajanja skript v kontekstu legitimne spletne aplikacije, kar predstavlja neposredno varnostno tveganje (OWASP, 2021).

2.3.3 Path Traversal

Path Traversal (imenovan tudi Directory Traversal) je varnostna ranljivost, ki napadalcu omogoča dostop do datotek in imenikov zunaj predvidenega direktorija aplikacije. Do ranljivosti pride, kadar aplikacija uporabniški vnos neposredno uporabi za določanje poti do datoteke brez ustrezne validacije ali omejitve dostopa. OWASP tovrstne napake uvršča med primere neustreznega nadzora dostopa, kjer aplikacija dostopa do sistemskih virov ne omeji pravilno (OWASP, 2021).

Mehanizem napada temelji na manipulaciji poti do datoteke z uporabo posebnih znakov, kot je zaporedje »../«, ki omogoča premikanje po strukturi direktorijev navzgor. Če aplikacija na primer omogoča prenos datoteke na osnovi parametra v URL-ju in ne preverja njegove vrednosti, lahko napadalec spremeni pot ter dostopa do občutljivih sistemskih datotek. Posledice lahko vključujejo razkritje konfiguracijskih datotek, dostop do podatkovnih baz ali drugih zaščitenih informacij, kar predstavlja resno varnostno tveganje (OWASP, 2021).

Primer napada:

Če aplikacija uporablja URL v obliki:

```
https://primer.si/download?file=dokument.txt
```

lahko napadalec spremeni parameter v:

```
https://primer.si/download?file=../../../../etc/passwd
```

V primeru neustreznega preverjanja vhodnih podatkov aplikacija omogoči dostop do sistemske datoteke, ki sicer ne bi smela biti javno dostopna.

2.4 Koncept varnega razvoja

2.4.1 Vključevanje varnosti v življenjski cikel razvoja programske opreme (SDLC)

Vključevanje varnosti v življenjski cikel razvoja programske opreme oziroma SDLC je danes eden izmed osnovnih pristopov pri razvoju zanesljivih informacijskih sistemov. V starejših razvojnih modelih je bila varnost pogosto obravnavana šele po implementaciji ali testiranju sistema. Tak način dela je pomenil, da so bile ranljivosti odkrite pozno, njihovo odpravljanje pa je bilo zato zahtevnejše in dražje. Sodobnejši pristopi zato izhajajo iz tega, da morajo biti varnostne zahteve upoštevane že pri oblikovanju sistema. S tem je povezano tudi načelo »Security by Design«, ki zagovarja, da se varnostni mehanizmi vključijo že v arhitekturo sistema (Sharma, 2025).

Tak pogled na razvoj je tesno povezan tudi z načelom »shift-left«. Gre za idejo, da se vprašanja varnosti ne prelagajo na konec razvoja, ampak se obravnavajo sproti skozi posamezne faze projekta. To vključuje definiranje zahtev, modeliranje groženj in načrtovanje arhitekture. Prednost takega pristopa je, da je mogoče tveganja prepoznati pravočasno, bolje oceniti njihove posledice in dosledneje spremljati varnostne zahteve skozi celoten razvojni proces. Hkrati so tudi poznejši popravki navadno manj zahtevni, odločanje v projektu pa bolj sledljivo (Sinan, Shahin in Gondal, 2025).

Pomemben del vključevanja varnosti v SDLC sta tudi avtomatizacija varnostnih postopkov in vzpostavljanje kulture skupne odgovornosti. Pri sodobnih pristopih, kot je DevSecOps, varnost ni več naloga samo ene specializirane ekipe. Postane del dela razvijalcev, operativnih ekip in varnostnih strokovnjakov. Pri tem imajo veliko vlogo avtomatizirana orodja, sprotno preverjanje skladnosti z varnostnimi zahtevami in stalno spremljanje tveganj (Sinan, Shahin in Gondal, 2025).

Sistematično vključevanje varnosti v SDLC zato predstavlja proaktiven način obvladovanja ranljivosti. Omogoča zgodnje zaznavanje pomanjkljivosti, zmanjšuje možnost resnih varnostnih incidentov ter prispeva k večji robustnosti in dolgoročni zanesljivosti programske opreme (Sharma, 2025; Sinan, Shahin in Gondal, 2025).

2.4.2 Varnostne prakse v fazi implementacije

Faza implementacije je ena izmed najbolj občutljivih točk v razvoju programske opreme. Takrat se načrtovane funkcionalnosti pretvorijo v dejansko izvorno kodo. Prav v tej fazi lahko nastanejo konkretne varnostne ranljivosti. Te so pogosto posledica slabega preverjanja vhodnih podatkov, neustreznega upravljanja dostopa ali napačne uporabe programskih konstrukcij. Zaradi tega sodobni pristopi varnega razvoja poudarjajo uporabo varnih programerskih praks. Sem sodijo dosledno preverjanje vhodov, pravilna obravnava napak in omejevanje privilegijev na najnižjo potrebno raven (Sinan, Shahin in Gondal, 2025).

Pomemben del varnega programiranja je tudi uporaba preverjenih knjižnic in ogrodij, ki že vsebujejo zaščitne mehanizme proti pogostim vrstam napadov. Poleg tega imajo pomembno vlogo ročni in avtomatizirani pregledi kode. Ti omogočajo, da se varnostne napake odkrijejo, še preden je koda vključena v produkcijsko okolje. Posledično se zmanjšuje možnost, da bi ranljivosti ostale v sistemu, hkrati pa to prispeva k boljši kakovosti programske opreme (Sinan, Shahin in Gondal, 2025).

Pri fazi implementacije je varnost najbolj povezana z dejanskimi odločitvami pri pisanju kode. Že majhna napaka v logiki programa, obdelavi vhodnih podatkov ali uporabi knjižnic lahko povzroči ranljivost, ki se kasneje prenese v delujoč sistem. Zato imajo v tej fazi veliko vrednost postopki, ki razvijalcu pomagajo pri sprotnem preverjanju kode. Sem sodijo pregledi kode, uporaba varnih programerskih pravil in avtomatizirana orodja, ki lahko opozorijo na težave še

pred nadaljnjim razvojem ali vključitvijo kode v produkcijsko okolje (Sharma, 2025; Sinan, Shahin in Gondal, 2025).

2.4.3 Preverjanje in zagotavljanje varnosti

2.4.3.1 Statična analiza kode (SAST)

Statična analiza kode oziroma SAST je način varnostnega preverjanja, pri katerem aplikacije ni treba zagnati. Pregled se izvaja nad izvorno kodo, zato je tak pristop tesno povezan z notranjo zgradbo aplikacije. Prav zaradi tega se SAST pogosto opisuje tudi kot white-box pristop. Takšna analiza pomaga pri pregledu kode še med razvojem, ko je mogoče morebitne pomanjkljivosti lažje obravnavati (Dencheva, 2022).

Pri statični analizi je v ospredju koda. Orodja iščejo ranljivosti in druge napake, ki bi lahko pozneje vplivale na varnost ali zanesljivost sistema. Zaradi tega ima SAST pomembno vlogo v sodobnih razvojnih postopkih, kjer je cilj sprotno preverjanje kakovosti in varnosti kode (Dencheva, 2022; Sinan, Shahin in Gondal, 2025).

2.4.3.2 Dinamično testiranje aplikacij (DAST)

Dinamično testiranje aplikacij oziroma DAST temelji na preverjanju aplikacije med njenim delovanjem. Za razliko od statične analize se pri tem ne pregleduje izvorna koda, ampak se spremlja obnašanje sistema od zunaj. Zaradi tega se DAST pogosto povezuje z black-box pristopom. Tak način testiranja je uporaben predvsem pri odkrivanju ranljivosti, ki pridejo do izraza šele med dejansko uporabo aplikacije (Dencheva, 2022).

Pri DAST je v ospredju pogled uporabnika ali napadalca, ki nima vpogleda v notranjo zgradbo sistema. Orodje preverja, kako se aplikacija odziva na vhodne podatke in različne zahteve med izvajanjem. S tem lahko zazna težave, ki so povezane z dejanskim delovanjem aplikacije v realnem okolju. Prednost takšnega pristopa je, da pokaže, kako ranljivosti vplivajo na delujoč sistem, slabost pa je, da so takšne težave pogosto vidne šele pozneje kot pri statični analizi (Dencheva, 2022).

2.4.3.3 Integracija varnostnega testiranja v CI/CD (Continuous Integration/Continuous Delivery)

Sodobni razvoj programske opreme temelji na tem, da se varnostne kontrole vključujejo neposredno v razvojni cevovod. V praksi to pomeni, da se orodja za statično in dinamično

analizo lahko izvajajo v procesih neprekinjene integracije in dostave. Na ta način je mogoče varnostno stanje aplikacije spremljati sproti in ne šele po zaključku razvoja. Tak pristop je skladen z načinom dela DevSecOps, kjer varnost postane del vsakodnevnega razvojnega procesa (Sinan, Shahin in Gondal, 2025).

Takšna integracija prinaša več prednosti. Ena izmed glavnih je, da se težave odkrijejo hitreje in da je odziv nanje lažje organizirati. Poleg tega avtomatizacija pomaga ohranjati stalno preverjanje skladnosti z varnostnimi zahtevami. To pride posebej prav v okoljih, kjer se spremembe kode izvajajo pogosto in kjer mora razvoj ostati hiter, ne da bi pri tem zanemarili varnost. S tem se vzpostavlja ravnotežje med agilnostjo razvoja in ustrezno ravnjo zaščite sistema (Sinan, Shahin in Gondal, 2025).

3 STATIČNA ANALIZA KODE (SAST)

3.1 Opredelitev statične analize kode

3.1.1 Definicija in osnovni namen statične analize (SAST)

Statična analiza programske kode predstavlja analitični pristop, pri katerem se programska rešitev proučuje brez njenega dejanskega izvajanja. Namen takšne analize je iz izvorne kode izluščiti informacije o strukturi, lastnostih in potencialnih napakah programa. Statična analiza temelji na uporabi algoritmov, ki pregledujejo programsko kodo kot statični artefakt in omogočajo pridobivanje dejstev o njenem delovanju. Takšen pristop omogoča sistematično razumevanje kode ter identifikacijo problematičnih delov že na ravni zapisa programa (Thomson, 2021).

V praksi se statična analiza uporablja za odkrivanje različnih vrst nepravilnosti, kot so nedosegljiva koda, uporaba nevarnih funkcij, nepravilni programski vzorci ter varnostne ranljivosti. S tem zgodnjim zaznavanjem varnostnih pomanjkljivosti prispeva k izboljšanju kakovosti in varnosti končne rešitve (Kuszczyński in Walkowski, 2023).

Pomembna lastnost statične analize je tudi to, da omogoča pregled različnih možnih poti izvajanja programa na osnovi njegove strukture. Dinamična analiza je pri tem bolj omejena, saj opazuje samo dejansko izvajanje programa v izbranem scenariju. Statična analiza zato ponuja širši vpogled v potencialno obnašanje programske rešitve. Ni vezana le na posamezne primere izvajanja, ampak obravnava programsko kodo kot celoto (Thomson, 2021).

3.1.2 Pristopi testiranja glede na dostop do kode: white-box, black-box, grey-box

Pri preverjanju programske opreme se uporablja več različnih pristopov testiranja. Ti se med seboj razlikujejo predvsem po tem, koliko vpogleda ima izvajalec testiranja v notranjo zgradbo programa. Med najbolj znane sodijo white-box, black-box in grey-box testiranje. Gre za tri različne načine, kako lahko preverjamo programsko kodo in njeno delovanje. Ker vsak pristop prikaže drug vidik sistema, se v praksi pogosto uporabljajo skupaj (Khan in Khan, 2012).

White-box testiranje temelji na popolnem vpogledu v notranjo strukturo programa. Oseba, zadolžena za testiranje, ima dostop do izvorne kode, logike in implementacije sistema. Zaradi tega lahko podrobno analizira poti izvajanja programa in išče napake, ki so povezane s strukturo kode ali načinom implementacije. Black-box testiranje deluje drugače. Pri njem testiralec nima vpogleda v notranje delovanje sistema, ampak preverja predvsem vhodne in izhodne podatke ter funkcionalnost programa. Grey-box testiranje pa predstavlja srednjo pot med obema pristopoma. Testiralec ima pri tem delni vpogled v notranjo zgradbo sistema, kar mu pomaga pri pripravi bolj ciljanih testnih primerov, hkrati pa še vedno ohranja pogled uporabnika (Khan in Khan, 2012).

3.1.3 Umestitev SAST med druge pristope testiranja (SAST vs. DAST)

Kot je bilo že predstavljeno v poglavju 2.4.3, statična in dinamična analiza predstavljata ključna pristopa k preverjanju varnosti programske opreme. Ponovimo, da se ti pristopi med seboj pomembno razlikujejo glede na način delovanja, fazo uporabe ter vrsto zaznanih ranljivosti. Statična analiza kode (SAST) se izvaja brez zagona aplikacije in omogoča pregled izvorne kode ter odkrivanje ranljivosti že v zgodnjih fazah razvoja, medtem ko dinamično testiranje (DAST) temelji na opazovanju obnašanja aplikacije med njenim izvajanjem (Sinan, Shahin in Gondal, 2025).

Ključna razlika med pristopoma se kaže v vrsti zaznanih ranljivosti in času njihovega odkrivanja. SAST omogoča zgodnje odkrivanje napak v izvorni kodi, kot so nepravilna validacija vhodnih podatkov ali nevarni programski vzorci, in tako prispeva k zmanjšanju stroškov odpravljanja napak v kasnejših fazah razvoja. Po drugi strani DAST omogoča zaznavanje ranljivosti, ki se pokažejo šele med dejanskim delovanjem sistema, na primer napake pri upravljanju sej ali nepravilne konfiguracije, ki jih iz kode ni mogoče vedno neposredno razbrati (Sinan, Shahin in Gondal, 2025).

3.2 Delovanje statične analize

3.2.1 Vhod in izhod statične analize

Pri statični analizi vhod predstavljajo programske datoteke, ki jih orodje vključi v pregled. To so lahko izvorna koda in drugi artefakti, povezani s programsko rešitvijo. Na tej osnovi orodje izvede analizo in poišče morebitne nepravilnosti, vzorce napak ter varnostne ranljivosti. Pri tem ne gre samo za iskanje posameznih problematičnih vrstic, ampak tudi za prepoznavanje širših značilnosti programa, kot so uporabljene funkcije, tokovi podatkov in odstopanja od vnaprej določenih pravil. Orodje iz vhodnih podatkov izlušči informacije, ki razvijalcu pomagajo pri razumevanju stanja kode (Kuszczyński in Walkowski, 2023).

Izhod analize je poročilo, ki vsebuje seznam zaznanih težav, njihovo lokacijo v kodi in oceno resnosti. V praksi takšno poročilo razvijalcu ne poda samo informacije, da je bila napaka zaznana, ampak tudi nakaže, kje v programu se nahaja in kako resna je lahko njena posledica. To poveča preglednost razumevanja ugotovitev in olajša določanje prioritet pri odpravljanju pomanjkljivosti. Orodja poskrbijo tudi za razvrščanje rezultatov glede na vrsto težave ali kršitev pravil, zato je končni izhod analize navadno dovolj strukturiran, da ga je mogoče uporabiti kot neposredno podporo pri nadaljnjem izboljševanju kode (Kuszczyński in Walkowski, 2023).

3.2.2 Tehnike analize kode

Pri statični analizi se uporabljajo različne tehnike, s katerimi orodje išče morebitne napake in varnostne ranljivosti v programski kodi. Ena izmed osnovnih tehnik je prepoznavanje vzorcev oziroma pattern matching. Pri tej tehniki se koda primerja z vnaprej določenimi vzorci, ki so povezani z znanimi napakami ali tveganimi programskimi rešitvami. To pri analizi zelo olajša iskanje pogostih in jasno prepoznavnih problemov v kodi (Kuszczyński in Walkowski, 2023).

Poleg tega se uporablja tudi pregled strukture programa. Tu je poudarek na razmerjih med posameznimi deli kode, na uporabi funkcij in na splošni organizaciji programa. Na tej ravni je lažje opaziti napake, ki niso vezane le na en sam ukaz ali eno vrstico, ampak izhajajo iz načina, kako je programska rešitev zgrajena (Kuszczyński in Walkowski, 2023).

Ker posamezna tehnika ne odkrije vseh vrst težav, orodja navadno združujejo več pristopov. Prav ta kombinacija poveča zanesljivost analize in razvijalcu poda uporabnejši pregled tveganih

delov programa. Rezultati so zato navadno širši in uporabnejši, kot če bi analiza temeljila samo na eni metodi (Kuszczyński in Walkowski, 2023).

3.2.3 Pravila in zaznavanje ranljivosti

Pri statični analizi imajo pomembno vlogo pravila, po katerih orodje presoja kodo. Ta pravila določajo, kateri vzorci, funkcije ali programske rešitve veljajo za tvegane. Nastanejo na osnovi znanih ranljivosti, pogostih programskih napak in neustreznih razvojnih praks. Ko orodje pregleduje kodo, išče ujemanja s takšnimi pravili in na tej osnovi označi mesta, kjer bi lahko prišlo do varnostne ali funkcionalne težave (Kuszczyński in Walkowski, 2023).

Tak sistem ni omejen samo na iskanje posameznih napak. Z njegovo pomočjo je mogoče zaznati tudi širše vzorce, ki so značilni za določene vrste ranljivosti. Sem sodijo nepravilna uporaba funkcij, napačno ravnanje z vhodnimi podatki ali druge tvegane rešitve. Pravila analizi dajejo jasna merila in pomagajo pri bolj enotni razvrstitvi ugotovitev, kar pomaga tudi pri primerjavi različnih orodij, saj so rezultati brez takšnega okvira težje primerljivi (Kuszczyński in Walkowski, 2023).

3.3 Prednosti in omejitve SAST

3.3.1 Prednosti SAST

Statična analiza kode prinaša bolj sistematičen pregled programske rešitve, saj temelji na vnaprej določenih pravilih. Zaradi tega je presoja kode bolj enotna in doslednejša. Orodja za statično analizo samodejno preverjajo skladnost z varnostnimi smernicami in iščejo odstopanja od priporočenih programskih praks. Razvijalec tako dobi jasnejši pregled nad stanjem kode in lažje opazi dele, ki zahtevajo dodatno preverjanje ali popravek (Kuszczyński in Walkowski, 2023).

Statična analiza ima prednost tudi v tem, da lahko v kratkem času pregleda večje količine kode. To je posebej koristno pri večjih in bolj zapletenih sistemih, kjer bi bil ročni pregled zelo zamuden. Ker analiza poteka avtomatizirano, je postopek bolj ponovljiv in manj odvisen od posameznega pregledovalca (Kuszczyński in Walkowski, 2023).

3.3.2 Omejitve SAST

Statična analiza kode ima tudi svoje omejitve. Ena izmed bolj znanih je pojav napačnih zaznav oziroma false positives. To pomeni, da orodje določen del kode označi kot problematičen, čeprav ta v praksi ne predstavlja dejanske ranljivosti. Takšni rezultati razvijalcu povečajo količino dela, saj mora dodatno preverjati, katere ugotovitve so res pomembne in katere ne. Če je takšnih zaznav preveč, analiza izgubi del svoje uporabne vrednosti in postane manj pregledna (Kuszczyński in Walkowski, 2023).

Druga omejitev je povezana s tem, da statična analiza ne zajame dejanskega izvajanja aplikacije. Zaradi tega ne pokaže vedno, kako se program obnaša v konkretnih okoliščinah. Nekatere ranljivosti se namreč pokažejo šele pri določenih vhodih, načinu uporabe ali v specifičnem okolju. Statična analiza je zato vezana predvsem na strukturo kode in na pravila, po katerih poteka pregled. Njena uspešnost je odvisna od kakovosti teh pravil in od tega, kako dobro orodje razume logiko programa. Posledično to pomeni, da sama po sebi ni dovolj za popolno varnostno oceno, jo je pa smiselno dopolniti z drugimi pristopi testiranja (Kuszczyński in Walkowski, 2023).

3.4 Metrike vrednotenja (Precision, Recall, F1)

3.4.1 Osnovni pojmi (TP, FP, TN, FN)

Za vrednotenje učinkovitosti metod, kot je statična analiza kode, se pogosto uporablja *matrika zmede* oziroma *confusion matrix*. Gre za tabelaričen prikaz razmerja med dejanskimi in napovedanimi rezultati klasifikacije. Ena dimenzija prikazuje dejansko stanje, druga pa napoved modela. Omogoča neposredno primerjavo med pravilnimi in napačnimi rezultati (Yang in Berdine, 2024).

Confusion matrix vsebuje štiri osnovne kategorije rezultatov. Pravilno pozitivni primeri oziroma true positive (TP) pomenijo, da model pravilno zazna prisotnost določene lastnosti, na primer ranljivosti v kodi. Napačno negativni primeri oziroma false negative (FN) nastanejo takrat, ko model te lastnosti ne zazna, čeprav je dejansko prisotna. Napačno pozitivni primeri oziroma false positive (FP) pomenijo, da model označi težavo, ki v resnici ne obstaja. Pravilno negativni primeri oziroma true negative (TN) pa označujejo primere, kjer model pravilno

ugotovi, da določena lastnost ni prisotna. Te štiri kategorije predstavljajo osnovo za izračun nadaljnjih metrik, kot so precision, recall in F1 (Yang in Berdine, 2024).

Tabela 1: Confusion matrix

		Napoved	
		Pozitivno	**Negativno**
Dejansko	**Pozitivno**	Pravilno pozitivno (TP)	Napačno negativno (FN)
	Negativno	Napačno pozitivno (FP)	Pravilno negativno (TN)

Vir: Lasten vir

3.4.2 Precision in Recall

Precision predstavlja metriko za vrednotenje uspešnosti klasifikacijskega sistema, ki meri delež pravilno pozitivnih napovedi med vsemi pozitivnimi napovedmi. Opredeljena je kot razmerje med številom pravilno pozitivnih primerov (TP) in vsoto pravilno pozitivnih ter napačno pozitivnih primerov (TP + FP) (Raschka, 2014; Zuva in Zuva, 2012).

Precision se izračuna z naslednjo enačbo:

$$Precision = \frac{TP}{TP + FP}$$

Ta metrika izraža, kolikšen delež napovedanih pozitivnih primerov je dejansko pravilen oziroma kako zanesljive so pozitivne napovedi modela. V kontekstu informacijskega poizvedovanja in klasifikacijskih sistemov precision predstavlja merilo natančnosti, saj upošteva razmerje med pravilno in napačno zaznanimi pozitivnimi primeri (Zuva in Zuva, 2012).

Recall predstavlja metriko, ki meri sposobnost sistema za zaznavanje dejansko pozitivnih primerov. Opredeljena je kot razmerje med številom pravilno pozitivnih primerov (TP) in vsoto pravilno pozitivnih ter napačno negativnih primerov (TP + FN) (Raschka, 2014; Zuva in Zuva, 2012).

Recall se izračuna z naslednjo enačbo:

$$Recall = \frac{TP}{TP + FN}$$

Recall izraža delež vseh dejansko pozitivnih primerov, ki jih sistem uspešno zazna. V literaturi je recall pogosto označen tudi kot true positive rate (TPR) ali sensitivity, saj meri sposobnost sistema, da prepozna relevantne primere v podatkovnem naboru (Raschka, 2014).

Metriki precision in recall skupaj omogočata celovitejše vrednotenje uspešnosti klasifikacijskega sistema. Precision se osredotoča na pravilnost pozitivnih napovedi, medtem ko recall meri sposobnost zaznavanja vseh dejansko prisotnih pozitivnih primerov (Zuva in Zuva, 2012).

3.4.3 F1

F1-mera predstavlja metriko za vrednotenje uspešnosti klasifikacijskega sistema, ki združuje metriki precision in recall v enotno vrednost. Opredeljena je kot harmonična sredina med precision in recall ter omogoča uravnoteženo oceno obeh metrik (Raschka, 2014).

F1-mera se izračuna z naslednjo enačbo:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

F1-mera temelji na harmonični sredini med precision in recall. Za razliko od navadne aritmetične sredine daje večjo težo nižjim vrednostim. To pomeni, da bo F1-mera visoka samo takrat, ko bosta visoka precision in recall. Če je eden izmed njiju nizek, se to hitro pokaže tudi pri končnem rezultatu (Raschka, 2014).

F1-mera zato združuje dva pomembna vidika uspešnosti klasifikacijskega sistema. Prvi je natančnost pozitivnih napovedi, ki jo opisuje precision, drugi pa sposobnost zaznavanja vseh pozitivnih primerov, ki jo opisuje recall. F1 se pogosto uporablja kot enoten kazalnik takrat, ko želimo obe metriki obravnavati skupaj in med njima ohraniti ravnotežje (Raschka, 2014).

Takšna mera je posebej uporabna v primerih, kjer uporaba precision ali recall ne poda dovolj celovite slike. F1 zato omogoča širši pogled na uspešnost sistema in je pri vrednotenju pogosto bolj informativna kot ločena uporaba samo ene izmed teh dveh metrik (Raschka, 2014).

4 PREDSTAVITEV ORODIJ ZA STATIČNO ANALIZO

4.1 SonarQube

SonarQube predstavlja širšo platformo za avtomatizirano preverjanje kakovosti in varnosti kode, ki vključuje več produktov in načinov uporabe. V tem diplomskem delu je v ospredju različica SonarQube Community Build (nedavno preimenovanje iz Community Edition oziroma CE), saj je bila uporabljena pri postavitvi testnega okolja in izvedbi primerjalne analize z orodjem Semgrep CE (SonarSource, 2026).

4.1.1 Namen orodja in vloga v procesu razvoja programske opreme

SonarQube CE je orodje za avtomatiziran pregled izvorne kode in statično analizo. Namenjeno je odkrivanju težav v kodi med razvojem programske opreme. Uradna dokumentacija ga opisuje kot lokalno nameščeno rešitev, ki preverja kodo na osnovi pravil. Pri tem se osredotoča na različne vidike kakovosti, med katere sodijo vzdrževanost, zanesljivost in varnost. To pomeni, da orodje ni uporabno samo za iskanje posameznih napak, ampak tudi za širši nadzor nad stanjem kode v projektu. Za to diplomsko delo je pomembno predvsem, da ga je mogoče namestiti v lastnem okolju in uporabiti pri neposredni primerjavi z orodjem Semgrep CE (SonarSource, 2026).

Pomembno je tudi, da SonarQube Community Build ni zamišljen kot orodje, ki ga razvijalec uporabi le enkrat ob koncu razvoja. Njegova vloga je precej širša. Dokumentacija navaja, da se vključuje v proces avtomatiziranega pregleda kode in da se lahko poveže s cevovodom CI/CD. To pomeni, da se analiza lahko izvaja redno in ne samo občasno. Na ta način je mogoče težave zaznati dovolj zgodaj, še preden se prenesejo v naslednje faze razvoja. Tak pristop je v praksi pomemben, ker je napake praviloma lažje odpraviti takrat, ko so spremembe še sveže in omejene na manjši del projekta. Community Build je pri tem usmerjen predvsem v analizo glavne veje projekta, medtem ko naprednejše možnosti analize več vej in zahtevkov za združevanje sprememb sodijo v druge različice SonarQube. Tudi to je za diplomsko delo pomembno, saj vpliva na način priprave testnega okolja in na zasnovo primerjave (SonarSource, 2026).

Vloga orodja v razvojnem procesu je povezana tudi z načinom, kako analiza dejansko poteka. Dokumentacija pojasnjuje, da se analiza izvede s pomočjo orodja SonarScanner. Ta pregleda

lokalni repozitorij in rezultate pošlje v SonarQube Community Build, kjer se nato obdelajo in prikažejo v obliki poročil. Pri tem imajo pomembno vlogo kakovostni profili. Ti določajo, katera pravila se uporabijo pri analizi posameznega programskega jezika. Vgrajeni profil Sonar way predstavlja začetno osnovo za večino projektov. Iz tega je razvidno, da SonarQube Community Build ne služi samo temu, da pokaže seznam zaznanih težav. Njegov namen je tudi v tem, da pomaga vzdrževati enotna pravila in standarde kakovosti znotraj projekta. Prav zato ga lahko obravnavamo kot značilen primer orodja SAST, ki je vključeno v dejanski razvojni potek dela in ne deluje ločeno od njega (SonarSource, 2026).

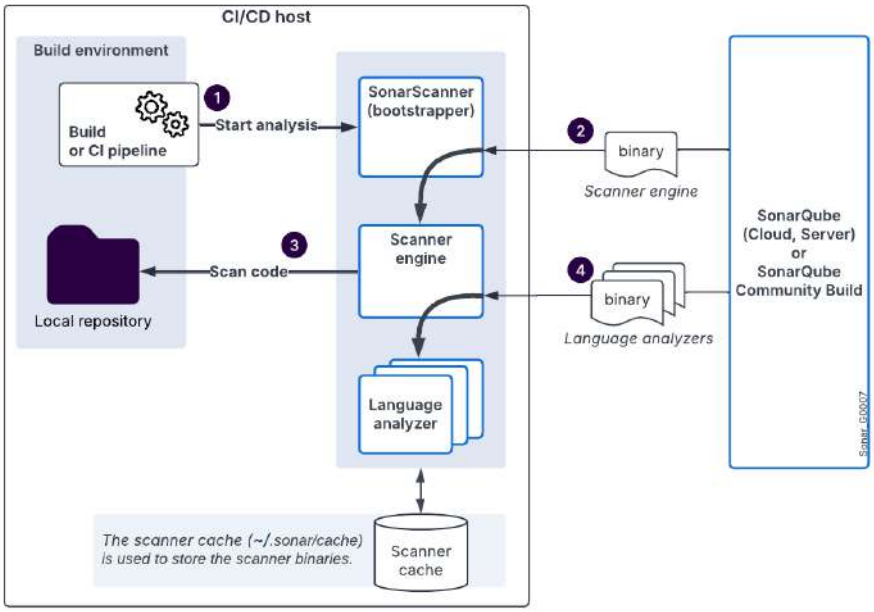
4.1.2 Način delovanja SonarQube Community Build pri analizi kode

Delovanje SonarQube Community Build temelji na avtomatizirani analizi izvorne kode, ki se izvaja kot del procesa pregleda kode. Dokumentacija pojasnjuje, da se analiza začne takrat, ko gradnja ali CI-cevovod zažene ustrezen SonarScanner. Ta nato pregleda lokalni repozitorij in glede na nastavljeni obseg analize določi, katere datoteke bodo vključene v pregled. Pomembno je tudi to, da se analiza ne izvaja neposredno na strežniku SonarQube, ampak na gostitelju CI/CD oziroma v okolju, kjer se projekt gradi ali preverja. SonarQube Community Build prejme rezultate analize, jih obdela ter iz njih pripravi poročila in izračuna kakovostna vrata. Ta zasnova pomeni, da je analiza ločena na del zajema podatkov in del njihove obdelave, kar omogoča avtomatizirano in ponovljivo preverjanje kode v razvojnem procesu (SonarSource, 2026).

Postopek analize poteka v več zaporednih korakih. Najprej SonarScanner pregleda lokalni repozitorij in določi datoteke, ki bodo analizirane. Nato pošlje zahtevo ustreznemu jezikovnemu analizatorju, ki datoteke analizira glede na uporabljeni kakovostni profil. Ko je analiza zaključena, analizator rezultate vrne skenerju, ta pa jih v obliki poročila posreduje v SonarQube Community Build. Sistem nato asinhrono obdela prejete podatke, prepozna nova odstopanja glede na nastavljene kakovostne standarde, izračuna kakovostna vrata in pripravi končna poročila. Dokumentacija pri tem posebej poudari, da se v projekt med analizo privzeto vključijo samo tiste datoteke, ki pripadajo jezikom, podprtim v uporabljeni izdaji. To je pomembno tudi za praktični del diplomskega dela, saj rezultat analize ni odvisen le od kode, ampak tudi od tega, katere jezike in datoteke izbrana izdaja sploh zna obravnavati (SonarSource, 2026).

Pomemben del delovanja SonarQube sta tudi izbira ustreznega SonarScannerja in priprava okolja za analizo. Dokumentacija navaja, da mora biti SonarScanner nameščen na gostitelju CI/CD, pri čemer je izbira odvisna od uporabljenega gradilnega sistema. Za projekte, zgrajene z Mavenom, Gradlom, .NET, NPM ali Pythonom, so na voljo namenski skenerji, za druge vrste projektov pa se uporablja SonarScanner CLI, ki zahteva več ročne nastavitve. Pred analizo je priporočena tudi ustrezna povezava z DevOps-platformo ali orodjem CI, prav tako pa mora biti pravilno nastavljena integracija s sistemom za upravljanje izvorne kode. Dokumentacija posebej opozarja, da je pri uporabi Gita potreben poln klon repozitorija, saj lahko plitek klon povzroči izpuščanje določenih informacij in vpliva na uspešnost analize. Iz tega je razvidno, da delovanje SonarQube Community Build ni odvisno samo od orodja, ampak tudi od pravilno pripravljenega okolja, v katerem se analiza izvaja (SonarSource, 2026).

Način delovanja SonarQube Community Build je povezan tudi s prilagoditvijo analize posameznemu projektu. Dokumentacija pojasnjuje, da se analiza upravlja s parametri, zapisanimi kot sonar lastnosti. Nekateri parametri so obvezni, številni drugi pa omogočajo prilagoditev obsega analize, vključitev testne pokritosti ali uvoz zaznav iz zunanjih analizatorjev. To pomeni, da SonarQube ne deluje po popolnoma fiksnem postopku, ampak omogoča določeno stopnjo prilagoditve glede na značilnosti projekta. V okviru diplomskega dela je to pomembno zato, ker lahko že nastavitev analize vpliva na končni rezultat in s tem tudi na primerjavo z orodjem Semgrep. Zato bo treba v praktičnem delu jasno opisati, kako je bila analiza nastavljena in kateri parametri so bili pri tem uporabljeni (SonarSource, 2026).



Slika 1: Potek analize kode v okolju SonarQube Community Build

Vir: (https://docs.sonarsource.com/sonarqube-community-build/analyzing-source-code/analysis-overview)

4.1.3 Pravila in kakovostni profili

SonarQube pri analizi kode temelji na pravilih. Ta pravila določajo, kaj orodje med pregledom išče in katere dele kode prepozna kot težavne. Če analiza zazna del kode, ki ni skladen z določenim pravilom, se to v rezultatih prikaže kot ugotovitev. Pravila so zato osnova celotnega delovanja SonarQube. Brez njih analiza ne bi mogla prepoznati, kaj v kodi odstopa od pričakovanih standardov. Njihova vloga ni samo v zaznavanju napak, ampak tudi v usmerjanju razvijalca k doslednejšemu in preglednejšemu pisanju kode (SonarSource, 2026).

Ker enak nabor pravil ni primeren za vsak projekt, SonarQube uporablja kakovostne profile. Kakovostni profil je zbirka pravil, ki so aktivna za določen programski jezik. To pomeni, da profil določa, katera pravila se bodo pri analizi dejansko uporabila. Za vsak jezik je lahko aktiven en profil, uporabnik pa lahko uporabi privzetega ali ga prilagodi svojim potrebam. Pomemben je tudi privzeti profil Sonar way, ki predstavlja začetno osnovo za večino projektov. V praksi to pomeni, da rezultat analize ni odvisen samo od kode, ampak tudi od tega, kako je analiza nastavljena in katera pravila so v profilu vključena (SonarSource, 2026).

Pomemben del sistema so tudi kakovostna vrata oziroma quality gates. Njihov namen je preveriti, ali projekt dosega določeno raven kakovosti. Če projekt izpolni postavljene pogoje,

kakovostna vrata prestane. Če jih ne, analiza to jasno pokaže. Tak mehanizem je uporaben predvsem takrat, ko analiza ni namenjena samo pregledu stanja kode, ampak tudi odločanju, ali je projekt pripravljen za naslednji korak v razvoju (SonarSource, 2026).

4.1.4 Podprti jeziki in pomen izbire jezika na analizo

SonarQube Community Build podpira analizo več različnih programskih jezikov. Med njimi so na primer Java, JavaScript, TypeScript, Python, PHP, C#, Go, Kotlin, Ruby, Rust, XML, YAML in še nekateri drugi. Poleg klasičnih programskih jezikov podpira tudi nekatere jezike in formate, povezane z infrastrukturo kot kodo, na primer Docker, Terraform, Kubernetes in CloudFormation. To je pomembno zato, ker SonarQube ni omejen samo na analizo ene vrste projektov, ampak ga je mogoče uporabiti v različnih razvojnih okoljih (SonarSource, 2026).

Pri izbiri jezika je treba upoštevati tudi to, da podpora ni povsem enaka pri vseh izdelkih in izdajah SonarQube. Na strani s podprtimi jeziki je jasno prikazano, da se nabor podprtih jezikov razlikuje med SonarQube Community Build, SonarQube Cloud in SonarQube Server. Nekateri jeziki so v Community Build na voljo, drugi pa sodijo v višje izdaje ali druge pakete. Zaradi tega izbira jezika neposredno vpliva na to, kaj bo orodje sploh lahko analiziralo in kako primerljivi bodo rezultati. Če bi bil za praktični del izbran jezik, ki v uporabljeni izdaji ni podprt, SonarQube takšne kode ne bi mogel obravnavati na enak način kot podprtega jezika. Zato mora biti v raziskavi jezik izbran premišljeno in usklajeno z dejanskimi zmožnostmi uporabljene različice orodja (SonarSource, 2026).

Pomembno je tudi to, da splošna tabela podprtih jezikov še ne pove vsega. SonarSource posebej navaja, da je treba za podprte različice jezika in za jezikovno specifične lastnosti pogledati posamezno stran izbranega jezika. To pomeni, da že oznaka, da je neki jezik podprt, še ne pomeni popolnoma enakega obsega analize pri vseh jezikih. V praksi je zato rezultat analize vedno povezan tudi s tem, kateri jezik je izbran in kako dobro ga SonarQube Community Build podpira (SonarSource, 2026).

4.1.5 Varnostne zmožnosti in omejitev obsega predstavitve

SonarQube Community Build ni namenjen samo preverjanju splošne kakovosti kode, ampak vključuje tudi varnostni vidik analize. To je vidno že pri razvrstitvi pravil, kjer so med glavnimi

področji navedeni varnost, zanesljivost, vzdrževanost in varnostna opozorila. To pomeni, da lahko orodje med pregledom kode zazna tudi takšne dele, ki so pomembni z vidika varnosti. Pri varnostni analizi ima posebno mesto pojem security hotspot. Gre za del kode, ki je z vidika varnosti občutljiv in zahteva dodatni pregled razvijalca. To še ne pomeni nujno, da gre za dejansko ranljivost. V določenih primerih se po pregledu izkaže, da težave ni, v drugih pa je potreben popravek. Tak pristop pokaže, da SonarQube ne temelji samo na samodejnem označevanju vseh spornih mest kot napak. Pri nekaterih ugotovitvah mora končno presojo opraviti človek. To je za to delo pomembno tudi zato, ker končni rezultat ni odvisen samo od orodja, ampak tudi od uporabnika, ki rezultate razume in jih zna pravilno presoditi (SonarSource, 2026).

Pri opisu varnostnih zmožnosti je treba hkrati jasno omejiti obseg predstavitve. SonarSource namreč posebej navaja, da je Advanced Security dodatek za višje izdaje SonarQube Server in da vključuje dodatne funkcije, na primer varnostno analizo, napredni SAST in analizo odvisnosti. Teh zmožnosti zato ne moremo pripisati različici Community Build, ki je uporabljena v tem diplomskem delu. Smiselno je torej predstaviti samo tiste varnostne funkcije, ki so dejansko del uporabljenega okolja (SonarSource, 2026).

4.1.6 Utemeljitev izbire SonarQube za praktični del diplomskega dela

Za praktični del diplomskega dela je bil izbran SonarQube CE, ker gre za brezplačno in prosto dostopno orodje za statično analizo kode. Takšna izbira je bila smiselna predvsem zato, ker omogoča postavitev lastnega testnega okolja brez dodatnih stroškov. Hkrati gre za uveljavljeno rešitev, saj SonarSource navaja, da SonarQube zaupa več kot sedem milijonov razvijalcev. Prav tako so priporočila za SonarQube pogosta na številnih programerskih forumih, kar me je tudi dokončno prepričalo (SonarSource, 2026).

Pri izbiri mi je bilo pomembno tudi, da SonarQube Community Build omogoča razmeroma enostaven začetek dela. Vgrajeni profil Sonar way je zasnovan tako, da je primeren za večino projektov kot začetna osnova. Zato se je za moje diplomsko delo, ki se osredotoča na začetniško primerjavo dveh rešitev SAST, zdel dobra izbira (SonarSource, 2026).

4.2 Semgrep

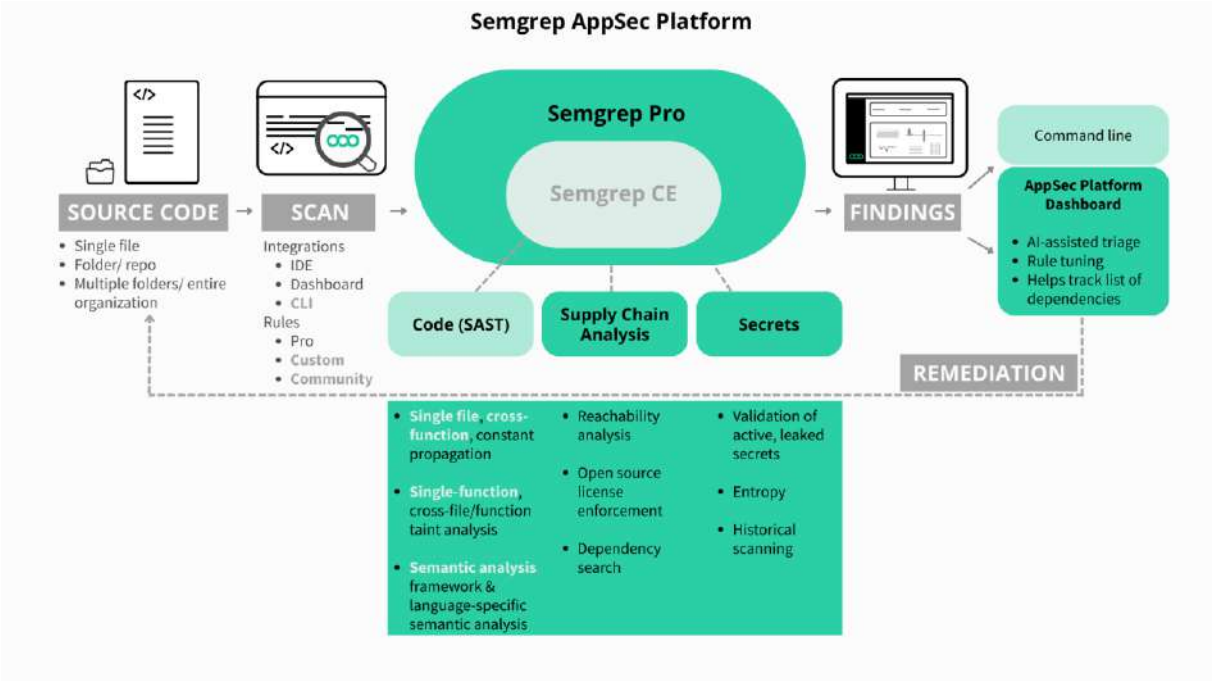
Semgrep CE predstavlja širšo platformo za analizo kode, ki vključuje več produktov in načinov uporabe. V tem diplomskem delu je v ospredju različica Semgrep CE, saj je bila uporabljena pri postavitvi testnega okolja in izvedbi primerjalne analize z orodjem SonarQube CE (Semgrep, 2026).

4.2.1 Namen orodja in vloga v procesu razvoja programske opreme

Semgrep CE je orodje za analizo kode, ki išče varnostne težave brez zagona programa. Njegov glavni namen je, da so morebitne ranljivosti in drugi sporni vzorci v kodi najdeni čim prej. Kot širši produkt Semgrep ne pokriva samo statične analize kode, ampak tudi analizo odvisnosti in odkrivanje občutljivih podatkov v kodi, kot so API-ključi, gesla in dostopni žetoni. Semgrep CE, ki je del tega diplomskega dela, pa je odprtokodna različica orodja, ki vključuje pogon za skeniranje kode, community pravila in vtičnike za razvojna okolja. Zaradi tega je Semgrep CE uporaben že v osnovni obliki in ni nujno vezan na širšo komercialno platformo (Semgrep, 2026).

Semgrep CE ima v procesu razvoja predvsem praktično vlogo. Uporabnik ga lahko zažene lokalno, v razvojnem okolju, pred oddajo kode ali kot del procesa CI/CD. To je koristno zato, ker se težave lahko odkrijejo dovolj zgodaj in ne šele na koncu razvoja. Orodje se zato ne uporablja samo za končni pregled projekta, ampak tudi kot pomoč med razvojem. Njegova velika prednost je tudi ta, da je lokalni zagon hiter in preprost. Posledično je Semgrep CE primeren za testno okolje, kjer želi uporabnik hitro videti prve rezultate in razumeti, kako orodje deluje (Semgrep, 2026).

Ena izmed bolj prepoznavnih lastnosti Semgrepa je način, kako uporablja pravila. Ta so zapisana v razmeroma preprosti obliki in se ujemajo s kodo na osnovi vzorcev. Uporabnik lahko uporabi že pripravljena pravila, pravila skupnosti ali napiše svoja. Prav v tem je njegova uporabna vrednost. Orodje ni omejeno samo na vnaprej pripravljene zaznave, ampak ga je mogoče prilagoditi projektu in načinu dela razvojne ekipe. Semgrep CE zato ni uporaben samo za osnovni varnostni pregled kode, ampak tudi za preverjanje pravil in vzorcev, ki so pomembni za posamezno okolje (Semgrep, 2026).



Slika 2: Predogled Semgrep AppSec platforme

Vir: (https://semgrep.dev/docs/introduction)

4.2.2 Način delovanja Semgrep CE pri analizi kode

Delovanje Semgrep CE temelji na tem, da orodje pri pregledu kode uporabi izbrana pravila in jih primerja z vsebino datotek v projektu. Pri lokalni uporabi se analiza najpogosteje zažene z ukazom *semgrep scan*. Tak zagon ne zahteva ločenega strežnika ali posebne postavitve okolja. Uporabnik namesti Semgrep CLI, izbere projekt in nato zažene pregled nad datotekami, ki jih želi analizirati. V tem smislu je začetek dela precej neposreden, ker se prvi pregled lahko izvede že nad lokalnim repozitorijem (Semgrep, 2026).

Potem analize je pri Semgrep CE tesno povezan s pravili. Ko uporabnik zažene pregled, orodje najprej naloži izbrani nabor pravil. Ta pravila lahko prihajajo iz Semgrep Registry, lahko pa jih uporabnik doda tudi iz lokalnega okolja. Vsako pravilo določa, katere vzorce v kodi je treba iskati. Ko se pravilo ujema z delom kode, Semgrep CE ustvari ugotovitev oziroma finding. Ta je lahko povezana z varnostjo, pravilnostjo delovanja, slogom pisanja kode ali drugimi vrstami težav. Način delovanja je močno vezan prav na pravila in na to, kako so ta napisana (Semgrep, 2026).

Uporabnik lahko zažene posamezno pravilo, širši nabor pravil ali kombinacijo več pravil hkrati. V dokumentaciji je jasno prikazano, da je mogoče uporabiti pripravljena pravila iz Registryja,

pa tudi lokalna pravila v obliki YAML. Zaradi tega je analiza lahko zelo splošna ali pa precej usmerjena v točno določeno vrsto problema. Če uporabnik spremeni nabor pravil, se spremeni tudi to, kaj bo Semgrep CE dejansko iskal in kaj bo vrnil med rezultati. Po končanem pregledu Semgrep CE izpiše povzetek analize, število zagnanih pravil, število pregledanih datotek in seznam zaznanih ugotovitev. Pri posamezni ugotovitvi so navadno vidni ime pravila, kratek opis težave in točna vrstica kode, kjer je bilo zaznano ujemanje. Tak izpis daje uporabniku precej neposreden vpogled v to, kaj je bilo najdeno in zakaj. Že iz izhoda je mogoče razbrati, katero pravilo je sprožilo zaznavo in na kateri del kode se ta nanaša (Semgrep, 2026).

Semgrep CE se lahko uporablja tudi v razvojnih cevovodih. Pri tem dokumentacija loči med običajnim lokalnim pregledom in uporabo ukaza Semgrep CI, ki je namenjen okolju CI. V takem načinu uporabe analiza ni več samo enkraten pregled lokalne kode, ampak del rednega preverjanja sprememb v projektu. To je pomembno tudi za primerjavo s SonarQube, ker pokaže, da Semgrep CE ni omejen samo na hiter lokalni pregled, ampak se lahko vključi tudi v bolj urejen razvojni proces. Primerjavo obeh orodij pa bomo podrobneje opisali v poglavju 5.2.3 (Semgrep, 2025).

4.2.3 Podprti jeziki in pomen izbire jezika za analizo

Semgrep CE podpira večje število programskih jezikov. Na strani s podprtimi jeziki je navedeno, da Semgrep Code podpira več kot 35 jezikov. Med splošno podprtimi jeziki so na primer Java, JavaScript, Python, C#, Go, Kotlin, PHP, Ruby, Scala, Swift in Rust. Poleg tega so navedeni tudi nekateri dodatni jeziki in formati, ki so še v eksperimentalni fazi, kot so Bash, Dockerfile, HTML in XML (Semgrep, 2026).

Za analizo ni pomembno samo to, ali je jezik na seznamu, ampak tudi, kako dobro je podprt. Semgrep CE jezike razvršča glede na stopnjo zrelosti in obseg funkcij, ki so pri posameznem jeziku na voljo. Pri nekaterih jezikih so na voljo naprednejše oblike analize, kot je cross-file dataflow analysis. Pri drugih je podpora bolj omejena. To pomeni, da dva projekta nista nujno analizirana na enaki ravni, čeprav sta oba napisana v jeziku, ki ga Semgrep CE uradno podpira (Semgrep, 2026).

To je za praktični del diplomskega dela precej pomembno. Če je izbran jezik dobro podprt, lahko uporabnik pričakuje širši in zanesljivejši pregled kode. Če je podpora manj razvita ali je jezik označen kot eksperimentalen, je treba rezultate razumeti previdneje. Izbira jezika

neposredno vpliva na obseg analize, globino zaznav in to, kako uporabni bodo končni rezultati (Semgrep, 2026).

4.2.4 Varnostne zmožnosti in omejitev obsega predstavitve

Semgrep CE je v osnovi varnostno usmerjeno orodje. Semgrep pa je kot širši produkt zasnovan precej širše od pregledovanja izvorne kode. V tem diplomskem delu je treba pogled zožiti na tisti del orodja, ki je bil dejansko uporabljen pri praktičnem delu (Semgrep, 2026).

Pri opisu varnostnih zmožnosti je pomembna ločitev med Semgrep CE in širšo Semgrep AppSec Platform. Dokumentacija jasno navaja, da je Community Edition odprtokodno orodje za statično analizo, ki vključuje skenirni pogon, community pravila in vtičnike za razvojna okolja. Hkrati pojasnjuje, da komercialna platforma doda še dodatne možnosti, kot so upravljani pregledi na ravni organizacije, napredna Pro pravila, analiza odvisnosti, odkrivanje občutljivih podatkov in z umetno inteligenco podprta obravnava rezultatov. Potemtakem ni pravilno, da bi vse varnostne funkcije širšega ekosistema Semgrep avtomatično pripisali tudi različici CE (Semgrep, 2026).

Za to poglavje je zato smiselno opisati predvsem tiste zmožnosti, ki so neposredno povezane s Semgrep CE. Sem spada predvsem statična analiza kode na osnovi pravil, usmerjena v iskanje varnostnih težav v izvorni kodi. Širše funkcije, kot sta analiza odvisnosti in odkrivanje občutljivih podatkov, so za razumevanje produkta sicer pomembne, vendar ne sodijo v jedro praktične primerjave, če v testnem okolju niso bile dejansko uporabljene. Zato sem se odločil omejiti opis teh funkcij, da bo predstavitev orodja ostala poštena in vezana na dejanski obseg uporabe v diplomskem delu (Semgrep, 2026).

4.2.5 Utemeljitev izbire Semgrep CE za praktični del diplomskega dela

Za praktični del diplomskega dela sem kot drugo orodje za statično analizo kode izbral Semgrep CE, ker gre za odprtokodno orodje, ki je prosto dostopno in ga je mogoče preizkusiti brez dodatnih stroškov, podobno kot SonarQube CE. Za začetek dela ni potreben ločen strežnik, saj je osnovni pregled mogoče izvesti lokalno z uporabo Semgrep CLI. To mi je bilo pri praktičnem delu pomembno predvsem zaradi preprostejše priprave testnega okolja in neposrednejšega prvega preizkusa nad lastno kodo (Semgrep, 2026).

Pred izbiro orodij za diplomsko delo sem v dokumentaciji opazil, da osnovni skener podpira več kot 30 programskih jezikov, kar se mi je zdelo dodatno spodbudno. Ob pregledu dokumentacije sem dobil občutek, da je Semgrep CE primeren za uporabnika, ki želi hitro začeti z analizo kode in pri tem ne potrebuje širše komercialne platforme. Za diplomsko delo, ki se osredotoča na začetniško primerjavo dveh rešitev SAST, se mi je zato izbira tega orodja zdela smiselna in praktična (Semgrep, 2026).

4.3 Primerjalni teoretični pregled orodij

4.3.1 Primerjava načina delovanja orodij

SonarQube CE in Semgrep CE se med seboj razlikujeta že v osnovnem načinu delovanja. SonarQube temelji na širšem sistemu, v katerem ima pomembno vlogo strežniški del orodja. Analiza se zažene s SonarScannerjem, rezultati pa se nato pošljejo v SonarQube, kjer se dodatno obdelajo in prikažejo v obliki poročil ter kakovostnih vrat. Semgrep CE deluje bolj neposredno. Uporabnik zažene pregled nad projektom, orodje naloži izbrana pravila in nato poišče ujemanja v kodi. Rezultati se po koncu pregleda izpišejo neposredno uporabniku (SonarSource, 2026; Semgrep, 2026).

Druga razlika je v središču analize. Pri SonarQube imajo pomembno vlogo skener, jezikovni analizatorji, kakovostni profili in strežniška obdelava rezultatov. Pri Semgrepu pa so v ospredju pravila, po katerih orodje pregleduje kodo. SonarQube deluje kot bolj centraliziran sistem za pregled kakovosti kode, Semgrep CE pa kot bolj neposredno orodje za pregled kode na osnovi izbranih vzorcev in pravil (SonarSource, 2026; Semgrep, 2026).

4.3.2 Primerjava pravil, nastavitev in prilagodljivosti analize

Pri obeh orodjih imajo pravila pomembno vlogo, vendar je način njihove uporabe drugačen. SonarQube CE analizo gradi okoli kakovostnih profilov, kjer so za posamezni jezik določena aktivna pravila. Semgrep CE je bolj neposredno vezan na pravila, saj uporabnik izbira ali zapisuje pravila, po katerih bo pregled potekal. V prvem primeru je več poudarka na upravljanju profilov znotraj sistema, v drugem pa na pravilu kot osnovni enoti analize (SonarSource, 2026; Semgrep, 2026).

Razlika je vidna tudi pri prilagodljivosti. SonarQube dopušča prilagajanje pravil, resnosti in profilov, vendar v bolj urejenem in centraliziranem okviru. Semgrep CE gre pri tem bolj v smer odprtega prilagajanja, saj lahko uporabnik poleg obstoječih pravil uporablja tudi lastna pravila v obliki YAML. Semgrep CE je tako bolj neposredno prilagodljiv na ravni analize, medtem ko SonarQube spremembe usmerja skozi profile in sistemska pravila za posamezni jezik (SonarSource, 2026; Semgrep, 2026).

4.3.3 Primerjava podprtih jezikov in obsega varnostnih zmožnosti

Obe orodji podpirata več programskih jezikov, vendar je pri obeh pomembno, da podpora še ne pomeni tudi enake globine analize. SonarQube CE jezike povezuje s kakovostnimi profili in pravili za posamezni jezik, Semgrep CE pa pri podprtih jezikih posebej izpostavlja tudi stopnjo zrelosti podpore in razlike med posameznimi funkcijami analize. Zato pri obeh orodjih ni dovolj pogledati samo seznama jezikov, ampak tudi to, kako obsežna je podpora pri izbranem jeziku (SonarSource, 2026; Semgrep, 2026).

Obe orodji vključujeta varnostni vidik analize, vendar je pri obeh treba ločiti med osnovno različico Community oziroma CE in širšimi funkcijami, ki sodijo v druge izdaje ali dodatne produkte. Teoretična primerjava ne sme temeljiti na vseh zmožnostih širšega ekosistema, ampak na tistih funkcijah, ki so dejansko del različic, obravnavanih v tem diplomskem delu (SonarSource, 2026; Semgrep, 2026).

4.3.4 Teoretična primernost orodij za začetnike

Na teoretični ravni predvidevam, da je SonarQube CE za začetnika verjetno prijaznejši pri razumevanju rezultatov, Semgrep CE pa pri začetku uporabe. SonarQube temelji na preglednem prikazu ugotovitev znotraj širšega sistema kakovosti kode, Semgrep CE pa omogoča zelo hiter prvi zagon, ker je osnovni pregled mogoče izvesti lokalno prek okolja CLI, brez ločenega strežniškega dela. Že iz dokumentacije je zato mogoče sklepati, da SonarQube bolj poudarja preglednost rezultatov, Semgrep CE pa hitrost in preprostost prvega pregleda. Vsekakor pa bomo ustreznost teh predvidevanj preverili pri praktični uporabi.

Razlika se kaže tudi pri nastavitvah. SonarQube začetniku ponudi vgrajeni profil Sonar way, ki služi kot začetna osnova za večino projektov. Pri Semgrepu je v ospredju uporaba pripravljenih

pravil ali hitri pregled z osnovno konfiguracijo, nato pa se analiza po potrebi razširi z dodatnimi pravili. To pomeni, da se SonarQube bolj naslanja na vnaprej urejen sistem nastavitev, Semgrep CE pa na hiter vstop in postopno prilagajanje pregleda. S teoretičnega vidika sta zato lahko obe orodji primerni za začetnika, vendar vsako na nekoliko drugačen način. Dejanski odgovor na ta predvidevanja pa bo pokazala praktična uporaba (SonarSource, 2026; Semgrep, 2026).

4.3.5 Stroškovna tabela obeh orodij

4.3.5.1 SonarQube

Tabela 2: Cene produktov SonarQuba

Različica	Tip rešitve	Kaj vključuje?	Cenovni model
SonarQube Community Build	Brezplačna, lokalno nameščena	Osnovna statična analiza kode, podpora za ključne jezike, analiza glavne veje, osnovni pregled kakovosti in varnosti kode	Brezplačno
SonarQube Server Developer Edition	Plačljiva, lokalno nameščena	Dodatni jeziki, analiza vej in pull requestov, naprednejše funkcije za razvojne ekipe	750 $/leto
SonarQube Server Enterprise Edition	Plačljiva, lokalno nameščena	Širše upravljanje na ravni organizacije, naprednejše varnostne in upravljavske funkcije, podpora večjim ekipam	Cena po ponudbi

SonarQube Server Data Center Edition	Plačljiva, lokalno nameščena	Namenjena zelo velikim okoljem, večja razširljivost, visoka razpoložljivost, podpora velikim organizacijam	Cena po ponudbi
SonarQube Cloud Free	Brezplačna, rešitev v oblaku	Osnovni dostop do SonarQube Cloud, primerno za manjše projekte in uvodno uporabo	Brezplačna
SonarQube Cloud Team	Plačljiva, rešitev v oblaku	Funkcije za ekipe in podjetja, brez lokalnega vzdrževanja infrastrukture	32 $/mesec
SonarQube Cloud Enterprise	Plačljiva, rešitev v oblaku	Razširljivost, upravljanje na ravni podjetja, večje organizacijske zmogljivosti	Cena po ponudbi

Vir: Lasten vir

4.3.5.2 Semgrep

Tabela 3: Cene produktov Semgrepa

Različica	Tip rešitve	Kaj vključuje?	Cenovni model
Semgrep Community Edition (CE)	Brezplačna, odprtokodna	Osnovni SAST-mehanizem, odprtokodna pravila iz Semgrep Registry, možnost pisanja lastnih pravil, podpora za več kot 30 jezikov	Brezplačno
Semgrep Free Edition	Brezplačna, platforma	Osnovni dostop do platforme Semgrep, skeniranje do 10 repozitorijev, do 10 contributorjev, Pro pravila s cross-file analizo, osnovna AI-podpora	Brezplačno
Semgrep Teams	Plačljiva, platforma	Pro pravila, AI-podprto odkrivanje, triage in remediation, SCM- in CI/CD-integracije, SSO, podpora ekipam	30 $/mesec za kontributorja
Semgrep Enterprise	Plačljiva, platforma	Napredne funkcije za večje organizacije, večja razširljivost, širše upravljanje in podpora	Cena po ponudbi

Vir: Lasten vir

5 PRIPRAVA TESTNEGA OKOLJA

Praktični del diplomskega dela sem razdelil na dva ločena sklopa. Prvi sklop temelji na uporabi testnega nabora OWASP Benchmark za jezik Java, drugi pa na lastno pripravljeni manjši Java aplikaciji. Razlog za to odločitev se mi je zdel smiseln, saj oba pristopa služita različnemu namenu in skupaj podata bolj celovito sliko primerjave obeh orodij.

Raziskava na testnem sklopu OWASP Benchmark je bila namenjena preverjanju hipotez H1 in H2, saj se nanašata na uspešnost zaznavanja ranljivosti, natančnost in lažno pozitivne rezultate na standardnih primerih. Hipotezo H3 bom lahko argumentiral že med namestitvijo in konfiguracijo obeh orodij ter ob prvem zagonu. Lastna Java aplikacija pa je bila uporabljena pri preverjanju hipoteze H4, kjer je v ospredju vključitev orodja v krajši simulirani razvojni cikel.

5.1 Opis testnega okolja

5.1.1 OWASP Benchmark – Java

Za prvi del praktične raziskave sem uporabil testni sklop OWASP Benchmark za jezik Java. Gre za vnaprej pripravljen nabor testnih primerov, namenjen preverjanju uspešnosti varnostnih orodij pri odkrivanju ranljivosti v izvorni kodi. Uporabil sem uradni projekt BenchmarkJava, ki je javno dostopen v okviru projekta OWASP Benchmark. Gre za izvedljivo Java spletno aplikacijo, pripravljeno prav za vrednotenje orodij za odkrivanje ranljivosti. Ta standardiziran test je zato primeren za primerjavo funkcionalnosti orodij na istem naboru primerov. Če bi ta del raziskave temeljil samo na lastni aplikaciji, bi bil nabor primerov precej manjši, rezultati pa bolj vezani na mojo lastno implementacijo. Zato mi služi kot bolj strukturirana osnova, na kateri sem lahko obe orodji preverjal pod primerljivimi pogoji (OWASP, 2026).

V tem delu raziskave sem spremljal, kako uspešno SonarQube CE in Semgrep CE zaznavata izbrane ranljivosti. Ni me zanimalo samo, ali je posamezna ranljivost zaznana, ampak tudi, kako natančne so zaznave. Pri analizi sem beležil pravilne in napačne rezultate, ki so kasneje služili kot osnova za primerjavo obeh orodij.

5.1.2 Lastna Java aplikacija

Za drugi del praktične raziskave sem pripravil lastno manjšo aplikacijo v programskem jeziku Java. Aplikacija je bila namenoma zasnovana preprosto in pregledno, saj cilj ni bil razviti obsežnega sistema, ampak pripraviti nadzorovan primer za praktično preverjanje delovanja obeh orodij.

Aplikacijo sem pripravil v okolju Visual Studio Code. Vanjo sem vključil izbrane varnostne ranljivosti, ki sem jih želel preverjati v praktičnem delu raziskave. Pri zasnovi sem pazil, da so bili primeri dovolj preprosti za razumevanje in dovolj jasni, da je bilo mogoče spremljati odziv orodij na posamezne tipe ranljivosti.

Poseben poudarek je bil tudi na tem, da aplikacija ni služila zgolj kot enkraten testni primer, ampak kot preprost razvojni primer, na katerem sem lahko opazoval potek dela z orodjem. Zato sem za posamezne ranljivosti predvidel tako ranljive kot tudi popravljene različice implementacije. Tako sem lahko primerjal stanje kode pred analizo in po izvedenih popravkih, predvsem z namenom preverjanja četrte hipoteze.

Nisem opazoval le končnega rezultata analize, temveč tudi praktični vidik uporabe obeh orodij. Pomembno mi je bilo, kako preprosta je uporaba, kako pregledni so rezultati in kako hitro lahko tudi manj izkušen uporabnik najde napake v kodi.

5.1.3 Razlog za uporabo obeh pristopov

Za praktični del raziskave sem uporabil oba pristopa, ker sem želel primerjavo orodij izvesti z dveh različnih vidikov. Testni sklop OWASP Benchmark mi je služil kot bolj strukturirana osnova za merjenje uspešnosti zaznavanja ranljivosti. Lastna Java aplikacija pa mi je dala bolj praktičen vpogled v uporabo obeh orodij pri delu na manjšem projektu.

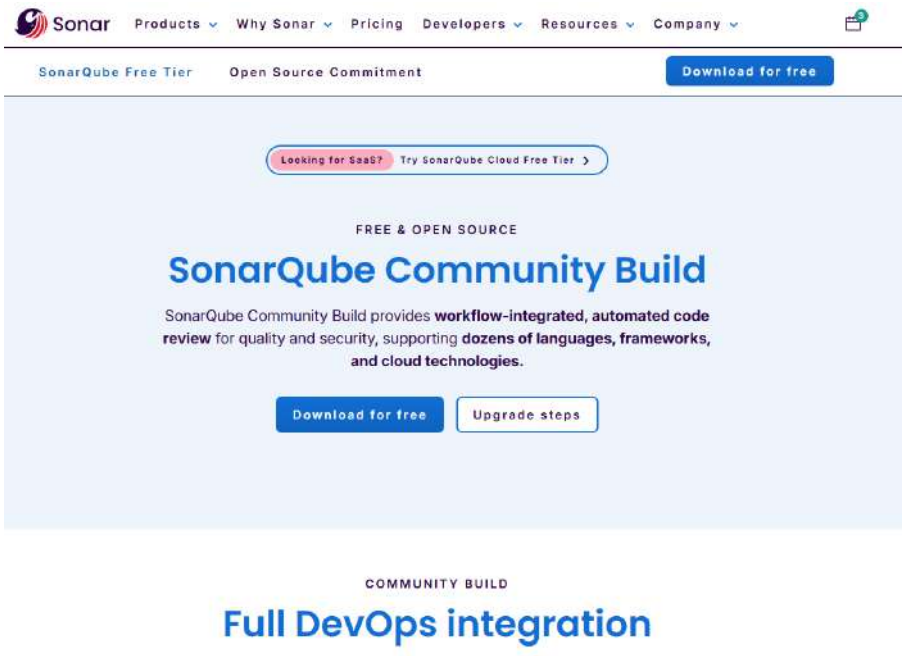
Samo eden od teh pristopov za celotno raziskavo ne bi bil dovolj. Če bi uporabil samo OWASP Benchmark, bi dobil predvsem merljiv del primerjave. Težje pa bi ocenil, kako orodji delujeta v bolj preprostem razvojnem primeru. Če bi uporabil samo lastno aplikacijo, bi bil praktični del bolj konkreten, vendar bi bila primerjava manj široka in bolj odvisna od moje lastne implementacije.

5.2 Priprava orodij za analizo

5.2.1 SonarQube CE

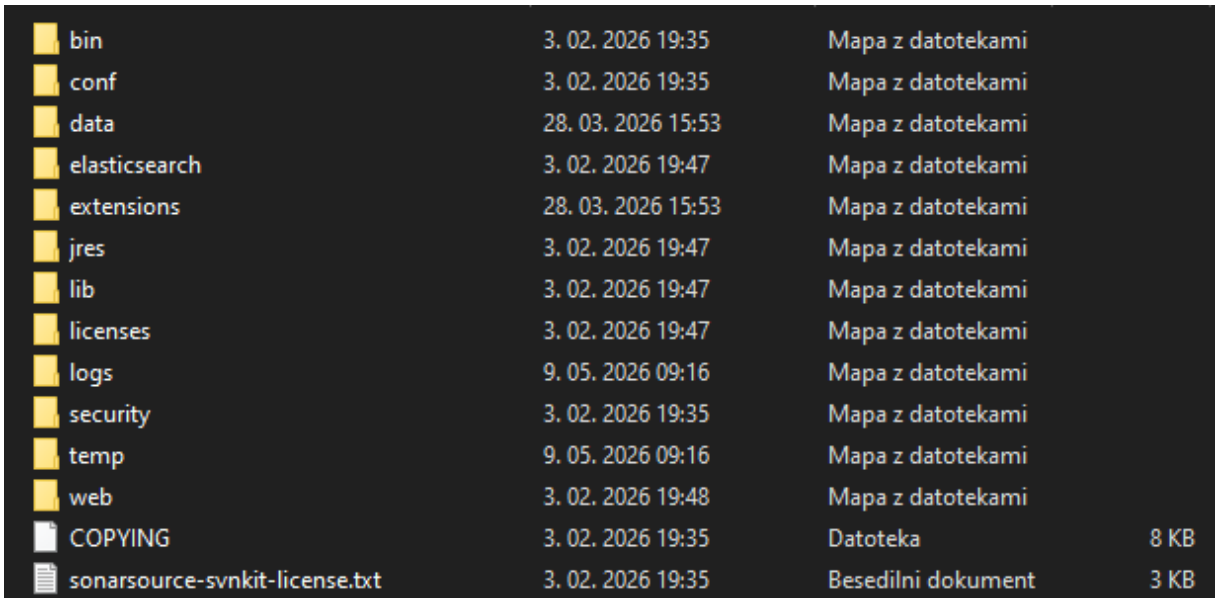
V tem delu raziskave sem opazoval, kako poteka začetna vzpostavitev SonarQube CE in katere korake je treba opraviti, da je orodje pripravljeno za uporabo pri analizi Java projektov.

SonarQube CE sem najprej prenesel z uradne spletne strani in ga razširil v lokalno mapo na računalniku. Ker gre za strežniško aplikacijo, je bilo za uporabo treba najprej zagnati lokalni SonarQube strežnik. V mojem primeru sem uporabljal operacijski sistem Windows, zato sem za zagon uporabil datoteko StartSonar.bat, ki se nahaja v mapi bin/windows-x86-64. Po zagonu je orodje v ukazni vrstici izpisalo potek inicializacije posameznih komponent, med katerimi sta najpomembnejša spletni strežnik in iskalni mehanizem Elasticsearch. Po uspešnem zagonu je bil spletni vmesnik dostopen na naslovu http://localhost:9000.



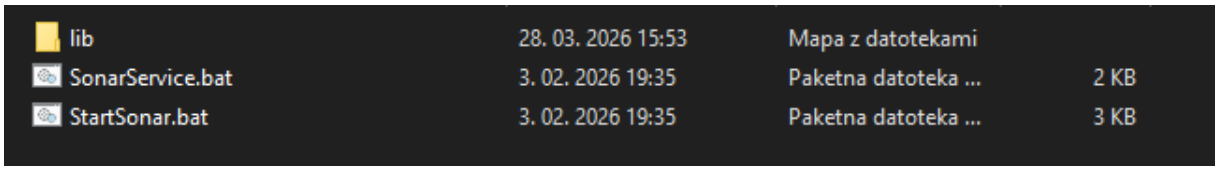
Slika 3: Stran za prenos SonarQube CE

Vir: (https://www.sonarsource.com/open-source-editions/sonarqube-community-edition/)



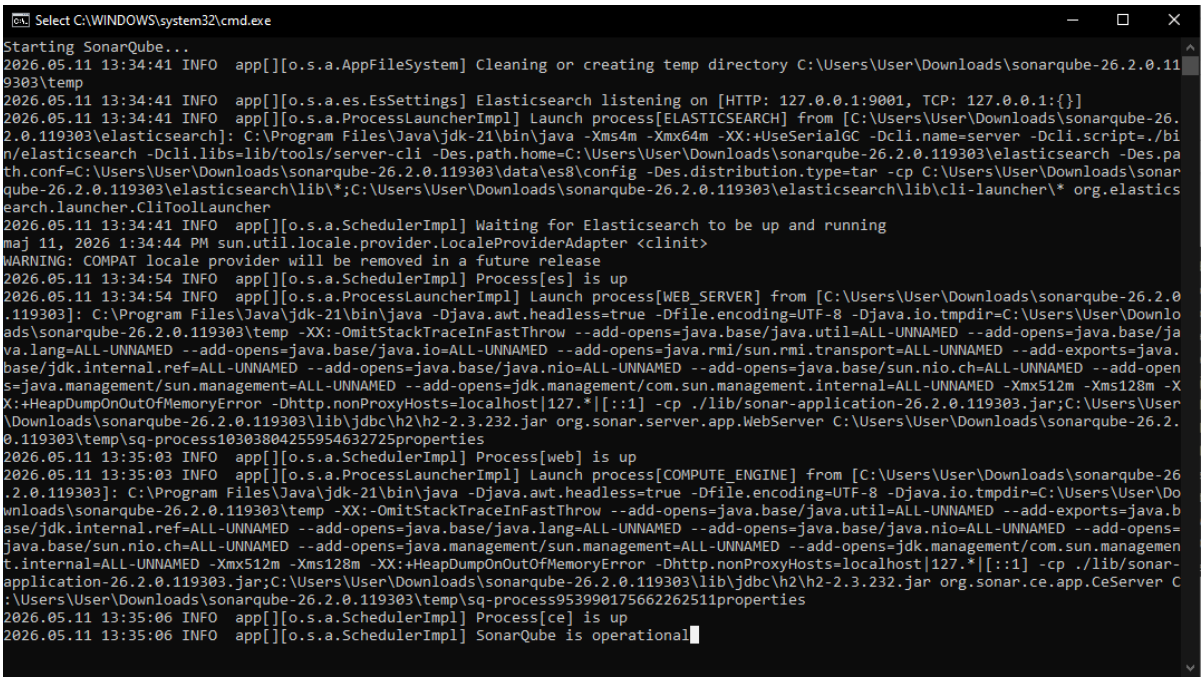
Slika 4: Lokalna mapa SonarQube

Vir: Lasten vir



Slika 5: Mapa bin/windows-x86-64

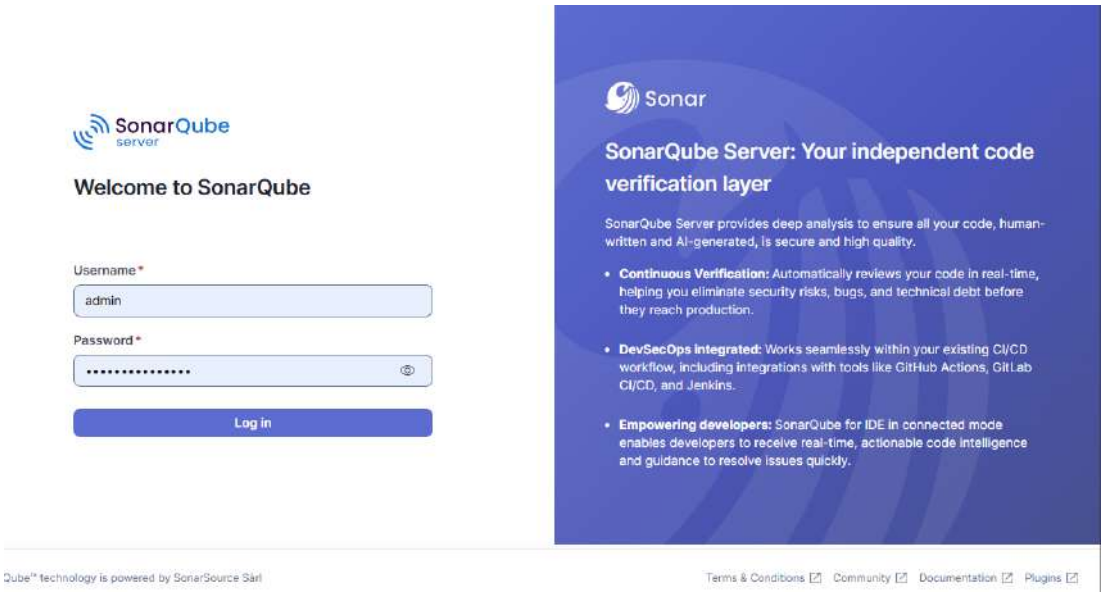
Vir: Lasten vir



Slika 6: Zagon SonarQube v ukazni vrstici po zagonu StartSonar.bat

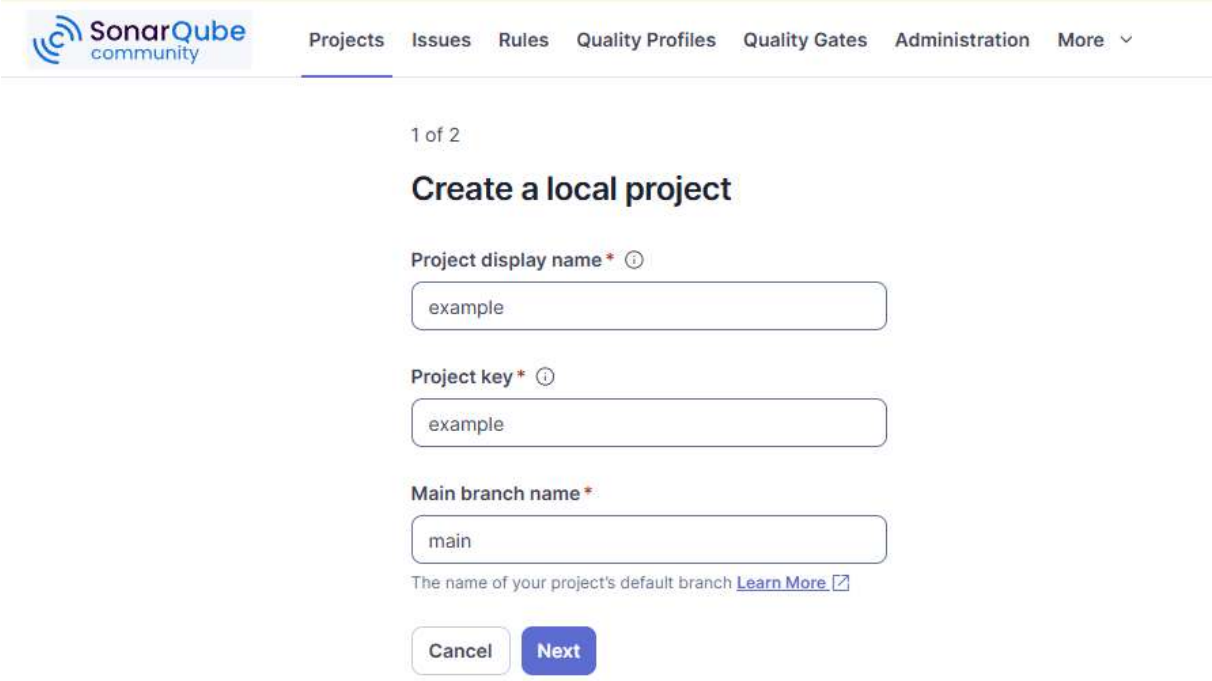
Vir: Lasten vir

Po uspešnem zagonu sem SonarQube odprl v spletnem brskalniku. Ob prvi prijavi sem uporabil privzete prijavne podatke, nato pa dostopal do začetnega uporabniškega vmesnika. Za nadaljnje delo je bilo treba ustvariti nov projekt, saj SonarQube rezultate analiz povezuje s posameznimi projekti. Pri ustvarjanju projekta sem določil ime projekta in njegov ključ. Ključ projekta je pomemben, ker se uporablja pri povezavi med lokalnim izvornim projektom in zapisom projekta v SonarQube.



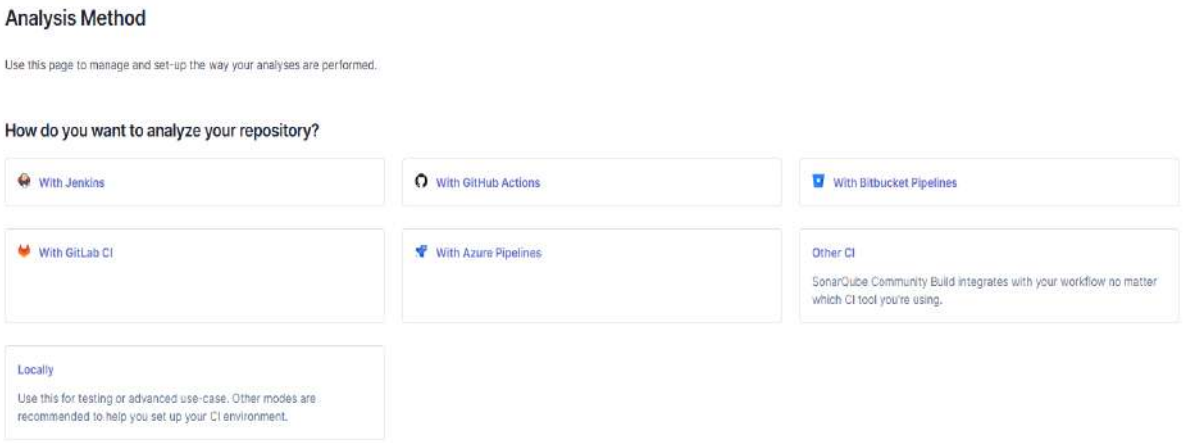
Slika 7: Prijavna stran SonarQube na naslovu http://localhost:9000

Vir: (http://localhost:9000/sessions/new?return_to=%2F)



Slika 8: Ustvarjanje novega projekta v vmesniku

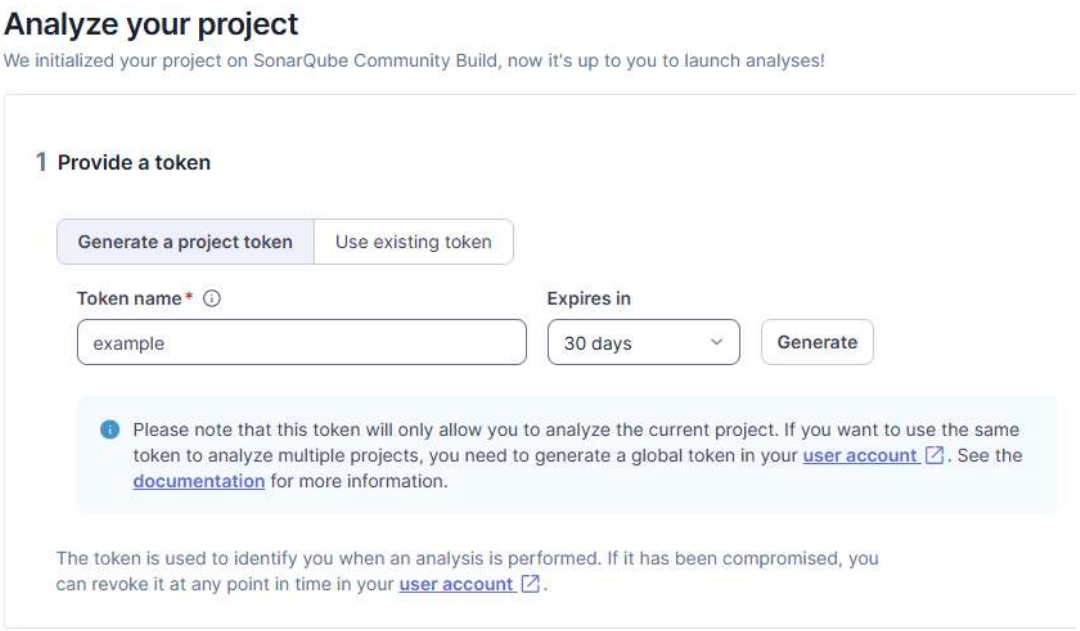
Vir: (http://localhost:9000/projects/create?mode=manual)



Slika 9: Izbira metode analize projekta

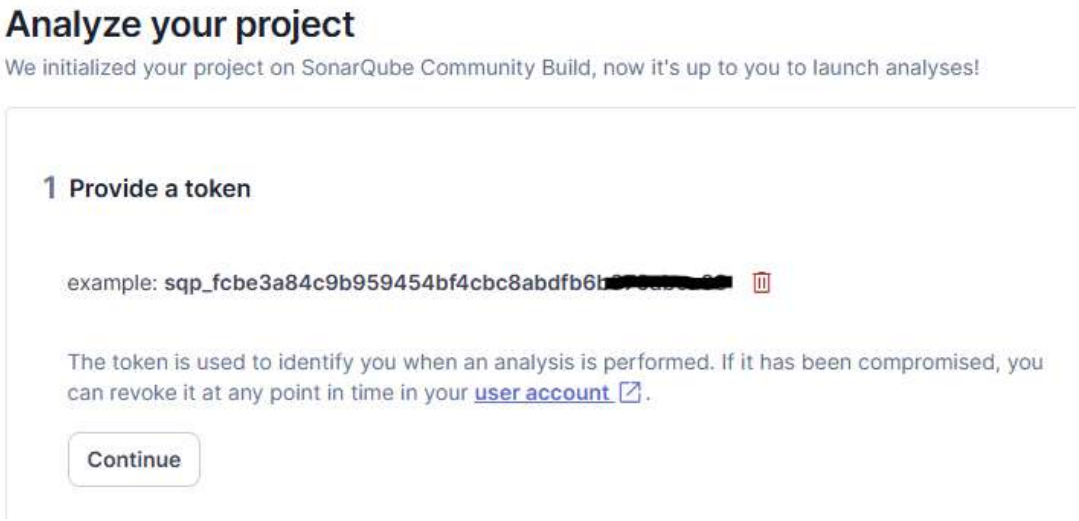
Vir: (http://localhost:9000/tutorials?id=example)

Po ustvarjanju projekta je bilo treba ustvariti še dostopni žeton oziroma token. Ta token omogoča, da lahko lokalno zagnano orodje SonarScanner pošlje rezultate analize na strežnik SonarQube. Generirani token sem shranil za nadaljnjo uporabo pri konfiguraciji analize. Ta korak je pomemben, ker brez pravilno nastavljenega tokena pošiljanje rezultatov ni mogoče. Zaradi varnosti tokena ni dovoljeno javno deliti, zato bo na sliki prekrit.



Slika 10: Generiranje tokena

Vir: (http://localhost:9000/tutorials?id=example&selectedTutorial=local)

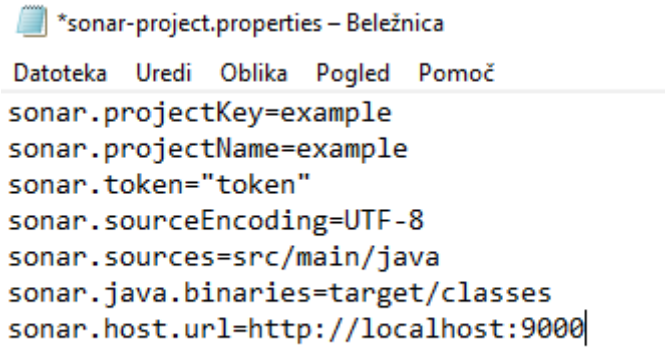


Slika 11: Token

Vir: (http://localhost:9000/tutorials?id=example&selectedTutorial=local)

Naslednji korak pri pripravi okolja SonarQube je bila konfiguracija povezave med lokalnim projektom in strežnikom SonarQube. To sem izvedel z datoteko sonar-project.properties, ki sem jo dodal v korensko mapo analiziranega projekta. V tej datoteki so določene osnovne nastavitve, kot so ključ projekta, ime projekta, naslov strežnika SonarQube, dostopni token, lokacija izvorne kode in lokacija prevedenih datotek. S tem sem pripravil vse potrebno za kasnejšo izvedbo analize.

Primer osnovne konfiguracije je prikazan spodaj:



Slika 12: Osnovna konfiguracija v datoteki sonar-project.properties v mapi projekta

Vir: Lasten vir

Pri pripravi orodja sem ugotovil, da vzpostavitev SonarQube zahteva več korakov kot nekatera preprostejša orodja za statično analizo. Poleg namestitve je treba zagnati lokalni strežnik,

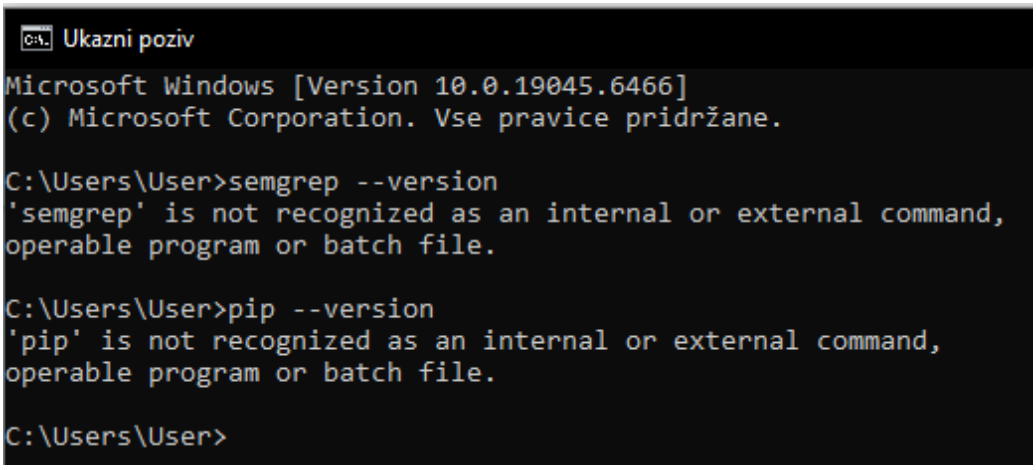
ustvariti projekt, pripraviti token in urediti konfiguracijo povezave z analizirano kodo. Pri Java projektih je treba poleg tega upoštevati tudi, da SonarQube za učinkovito analizo potrebuje prevedene .class datoteke. Zaradi tega je treba projekt pred dejansko analizo tudi ustrezno prevesti.

Na tej točki je bilo orodje SonarQube CE uspešno pripravljeno za uporabo v praktičnem delu raziskave. Nadaljnji koraki, povezani s konfiguracijo Semgrep CE, dejansko analizo OWASP Benchmarka in lastne Java aplikacije, so predstavljeni v nadaljevanju.

5.2.2 Semgrep

V tem delu raziskave proučujem, kako poteka začetna namestitev orodja in kako hitro ga je mogoče pripraviti za prvo uporabo v okolju Windows.

Pred namestitvijo sem najprej preveril, ali je bilo orodje v sistemu že dostopno iz ukazne vrstice. Pri tem sem ugotovil, da ukazov semgrep in pip v mojem okolju ni bilo mogoče klicati neposredno, saj nista bila dodana v sistemsko spremenljivko PATH. To pomeni, da Semgrep CE v začetnem stanju ni bil neposredno dostopen z običajnim ukazom.

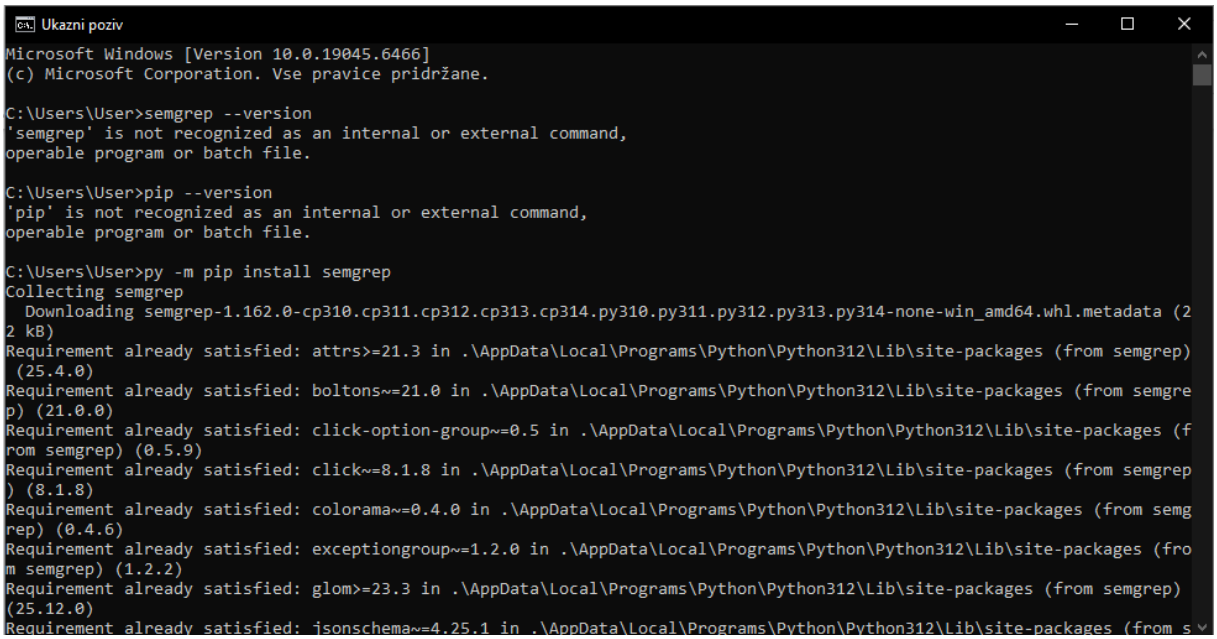


Slika 13: Ukaza semgrep --version in pip –version nista prepoznana

Vir: Lasten vir

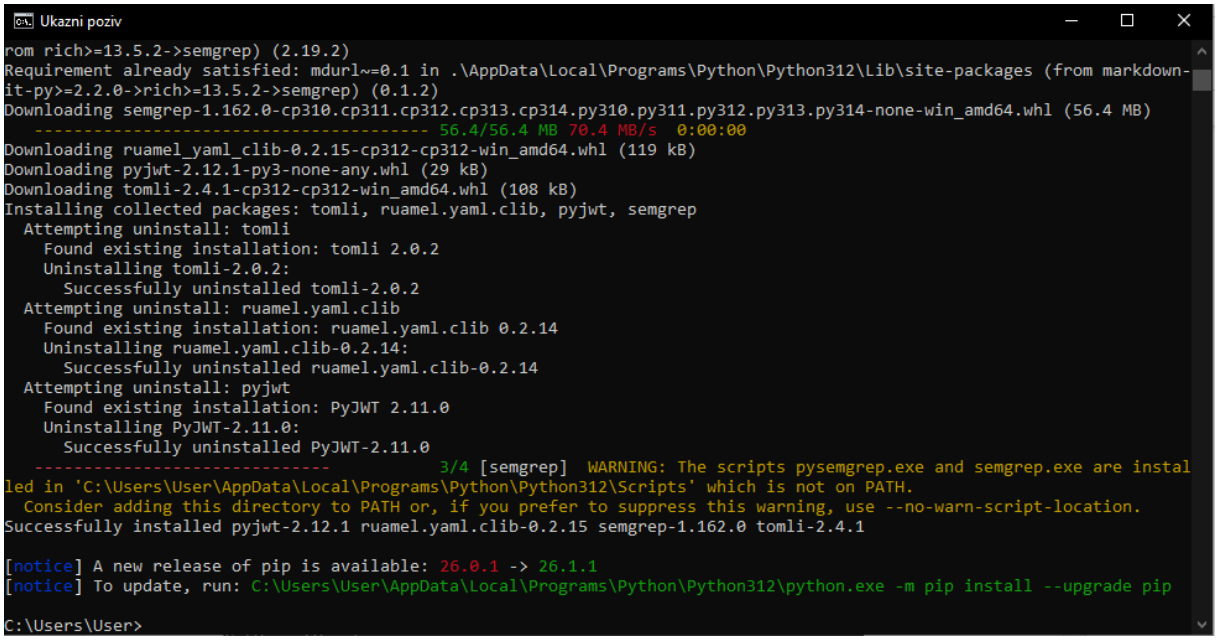
Ker neposreden dostop do ukazov ni deloval, sem Semgrep CE namestil z uporabo Pythonovega upravljalnika paketov. Namestitev sem izvedel z ukazom py -m pip install semgrep. Ta način

uporabe se je izkazal za zanesljivejšega, saj uporablja Pythonovo lokalno okolje ne glede na to, ali je pot do pip pravilno nastavljena v sistemu Windows.



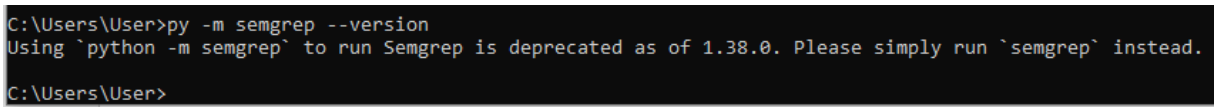
Slika 14: Začetek namestitve orodja Semgrep

Vir: Lasten vir



Slika 15: Uspešna namestitev orodja Semgrep

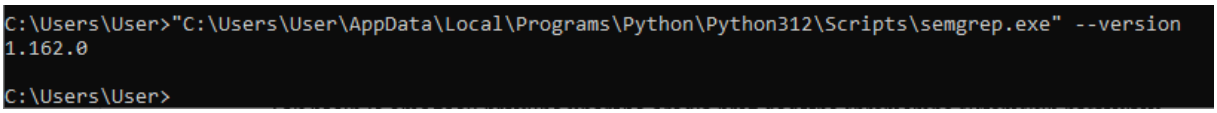
Vir: Lasten vir



Slika 16: Opozorilo pri preverjanju različice Semgrepa s py -m

Vir: Lasten vir

Semgrep CE sem zagnal z navedbo celotne poti do izvršljive datoteke semgrep.exe. Izpis različice je potrdil, da je bil Semgrep CE uspešno nameščen in pripravljen za uporabo.



Slika 17: Uspešen izpis različice Semgrepa

Vir: Lasten vir

Za analizo projekta v programskem jeziku Java Semgrep CE uporablja izbrani nabor pravil, ki se ob zagonu določi z ustreznim parametrom. Uporabil sem nabor pravil p/java, kot je prikazano na spodnji sliki, znotraj ukaza za zagon analize nad projektom.



Slika 18: Primer ukaza za analizo Java projekta s Semgrepom

Vir: Lasten vir

V mojem primeru je ta ukaz pomenil, da Semgrep CE uporabi pravila za jezik Java in pregleda izbrano mapo projekta. Za razliko od SonarQube CE pri Semgrep CE ni bilo treba ustvarjati posebnega projekta, pripravljati spletnega vmesnika ali zaganjati lokalnega strežnika. Posledično je bila začetna priprava orodja krajša in bolj preprosta. Na tej točki je bilo orodje Semgrep CE uspešno pripravljeno za uporabo v praktičnem delu raziskave.

5.2.3 Primerjava priprave obeh orodij za analizo

Pri pripravi praktičnega dela sem primerjal tudi zahtevnost namestitve, začetne konfiguracije in prve uporabe obeh orodij. Ta del primerjave je bil namenjen predvsem argumentiranju hipoteze H3, saj se nanaša na vprašanje, katero orodje je za začetnika hitreje vključiti v delo in pri katerem so zaznave bolj razumljive.

Pri orodju SonarQube CE je bila priprava okolja daljša in zahtevnejša. Treba je bilo vzpostaviti lokalni strežnik, ustvariti projekt, pripraviti dostopni žeton, urediti konfiguracijsko datoteko in nato ločeno zagnati analizo z orodjem SonarScanner. Poleg tega je bilo rezultate treba pregledovati v spletnem vmesniku, pri čemer sem se med pripravo srečal tudi z nekaterimi tehničnimi težavami pri zagonu in delovanju okolja. Za začetnika to pomeni več korakov in več možnosti za zaplete že pred prvo uspešno analizo.

Pri orodju Semgrep CE je bil postopek preprostejši. Orodje sem namestil v Pythonovo okolje in ga nato uporabljal neposredno iz ukazne vrstice. Za prvo analizo ni bilo treba vzpostavljati ločenega strežnika, ustvarjati projekta ali pripravljati dodatnih povezav med več komponentami. Tudi izpis rezultatov je bil bolj neposreden, saj je orodje pri zaznavah jasno navedlo tip problema, datoteko in mesto v kodi. Zaradi tega je bil pregled rezultatov hitrejši in bolj razumljiv.

Na osnovi praktične priprave imam dovolj izkušenj in znanja, da argumentiram tretjo hipotezo. Semgrep CE je za začetnika hitreje nameščen, preprosteje vključen v delo in bolj pregleden pri razlagi zaznav kot SonarQube CE.

Zato lahko na osnovi svojih izkušenj hipotezo H3 v okviru tega diplomskega dela **potrdim**.

6 IZVEDBA ANALIZE

6.1 Analiza na OWASP Benchmark

6.1.1 Priprava projekta za analizo

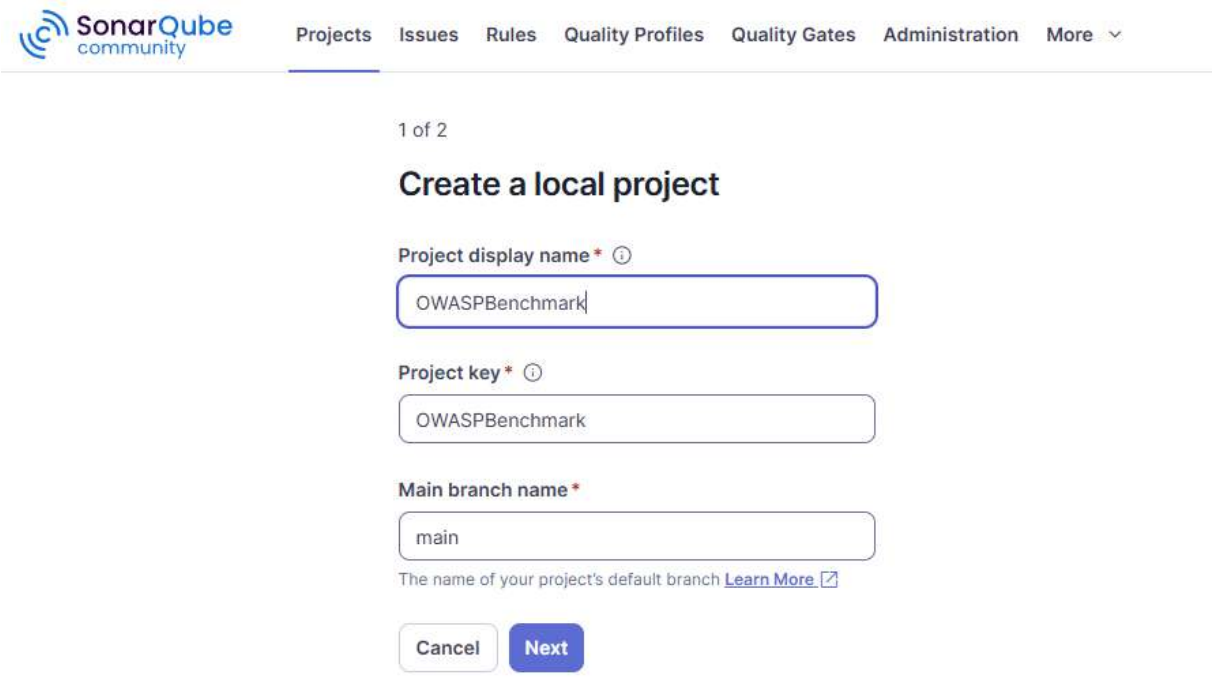
Za izvedbo praktične analize sem najprej pripravil projekt OWASP Benchmark za jezik Java. Pri tem sem izhajal iz uradnega projekta BenchmarkJava, ki sem ga uporabil kot standardizirano osnovo za primerjavo izbranih orodij. Prenesel sem ga s spletnega portala GitHub na povezavi https://github.com/OWASP-Benchmark/BenchmarkJava. Pridobil sem ga v obliki datoteke .zip, jo razširil v lokalno mapo ter projekt odprl v Visual Studio Code, kjer je bil pripravljen za nadaljnjo obdelavo.

6.1.2 Izvedba analize z orodjem SonarQube CE

Analizo projekta OWASP Benchmark sem najprej izvedel z orodjem SonarQube CE. Kot sem opisal v poglavju 5.2.1, gre za orodje, ki deluje kot lokalni strežnik s spletnim uporabniškim vmesnikom. Zato je treba pred dejansko analizo najprej zagnati okolje SonarQube CE in pripraviti povezavo med lokalnim projektom ter ustvarjenim projektom v vmesniku SonarQube CE.

Prvi korak je bil zagon lokalnega strežnika SonarQube. Kot sem opisal prej, to storimo z zagonom datoteke StartSonar.bat, ki se nahaja v mapi bin/windows-x86-64 znotraj namestitvene mape SonarQube. Po zagonu se je v ukazni vrstici začel postopek inicializacije, pri katerem sta se zagnala spletni strežnik in iskalni mehanizem Elasticsearch. Ko se je postopek uspešno zaključil, je bil SonarQube dostopen prek spletnega brskalnika na naslovu http://localhost:9000.

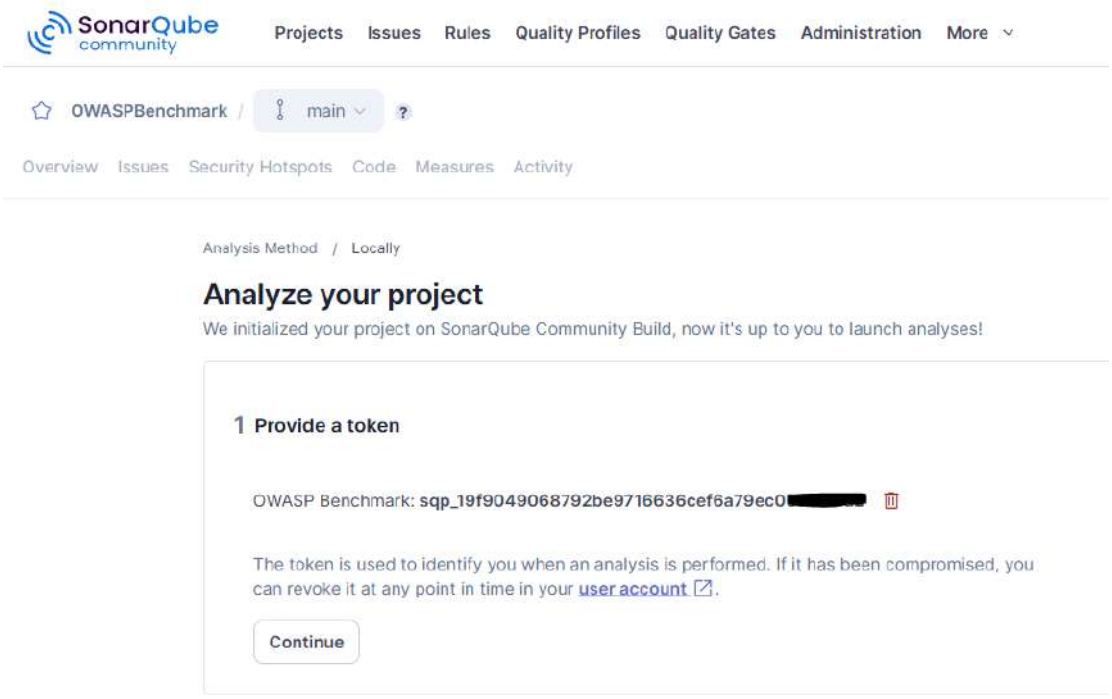
Po uspešnem zagonu sem v spletnem brskalniku odprl lokalni vmesnik SonarQube in se prijavil v sistem. Ker sem želel rezultate analize OWASP Benchmarka ločiti od drugih analiz, sem ustvaril poseben projekt za ta namen. Pri tem sem določil ime projekta in ključ projekta, ki se kasneje uporablja za povezavo z lokalno konfiguracijo analiziranega projekta. Na ta način sem zagotovil, da so bile ugotovitve analize povezane s pravilnim projektom in pregledno ločene od drugih testov.



Slika 19: Ustvarjanje projekta OWASP Benchmark

Vir: Lasten vir

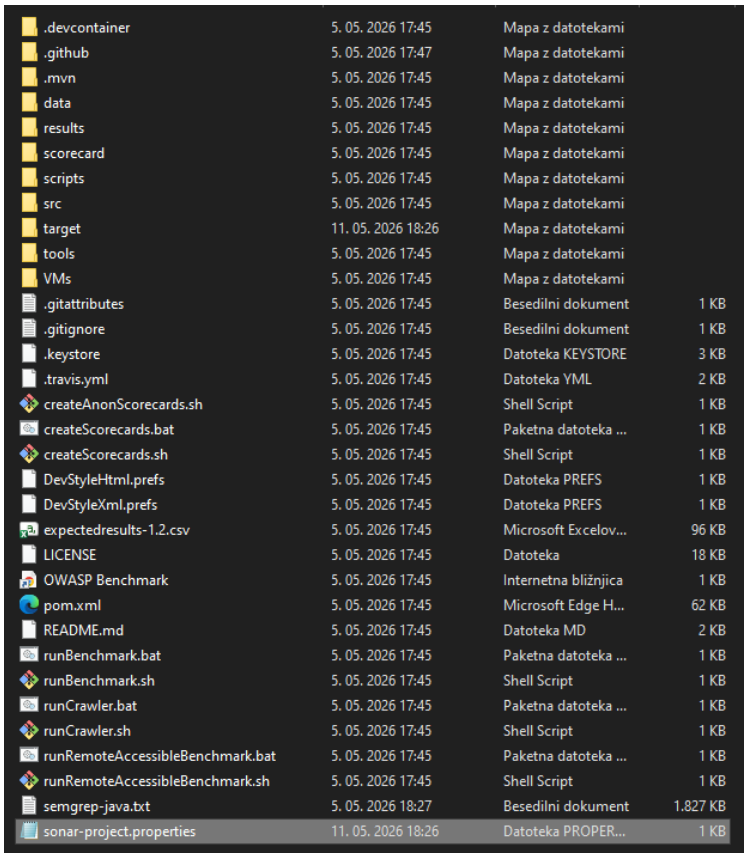
Po ustvarjanju projekta je bilo treba ustvariti še dostopni žeton oziroma token, ki ga SonarScanner uporablja za avtentikacijo pri pošiljanju rezultatov analize na strežnik SonarQube. Token sem ustvaril znotraj nastavitev projekta in ga uporabil v konfiguracijski datoteki lokalnega projekta. Zaradi varnosti dejanske vrednosti tokena nisem javno prikazal.



Slika 20: Pridobitev tokena za projekt

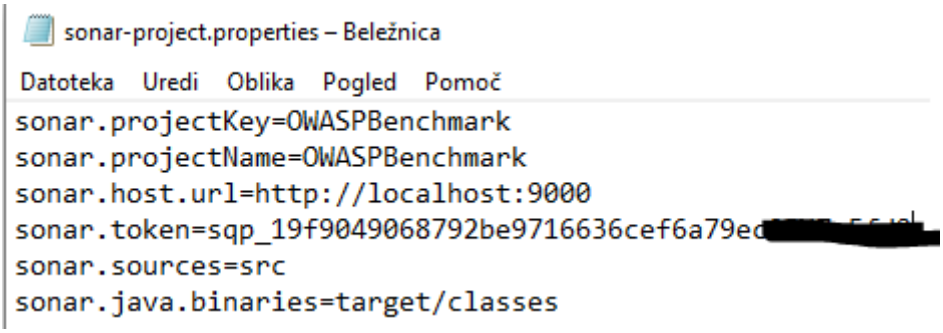
Vir: Lasten vir

Naslednji korak je bil, da sem v korensko mapo projekta OWASP Benchmark dodal datoteko sonar-project.properties, v kateri sem določil osnovne nastavitve za povezavo med lokalnim projektom in strežnikom SonarQube. V tej datoteki sem določil osnovne parametre, potrebne za analizo, kot so ključ projekta, ime projekta, naslov lokalnega strežnika SonarQube, dostopni token, lokacija izvorne kode in lokacija prevedenih Java razredov. Ta datoteka je predstavljala povezavo med lokalno kodo in projektom SonarQube.



Slika 21: Datoteka sonar-project.properties v korenski mapi projekta

Vir: Lasten vir



Slika 22: Konfiguracija datoteke sonar-project.properties za naš projekt

Vir: Lasten vir

Ker gre za projekt v programskem jeziku Java, je bilo pred analizo treba zagotoviti tudi uspešno prevajanje izvorne kode. To je bilo še posebej pomembno pri uporabi orodja SonarQube CE, saj to pri analizi Java projektov uporablja prevedene .class datoteke. Projekt sem zato najprej

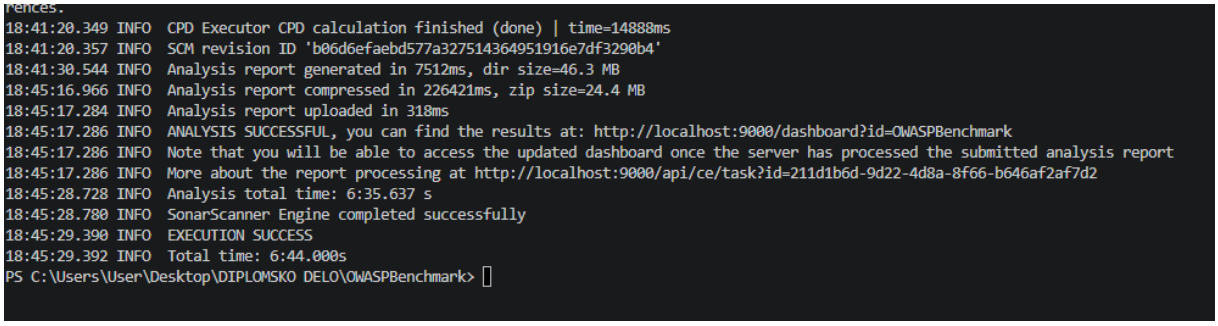
prevedel z uporabo orodja Maven in preveril, da se je postopek uspešno zaključil. To sem storil tako, da sem v razvojnem okolju odprl terminal in vpisal ukaz mvn clean package.

```
PS C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark> mvn clean package
[INFO] BuildTimeEventSpy is registered.
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< org.owasp:benchmark >------------------------
[INFO] Building OWASP Benchmark for Java 1.2
[INFO]   from pom.xml
[INFO] --------------------------------[ war ]---------------------------------
[INFO]
[INFO] --- clean:3.5.0:clean (default-clean) @ benchmark ---
[INFO] Deleting C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark\target
[INFO]
[INFO] --- enforcer:3.6.2:enforce (enforce-bytecode-version) @ benchmark ---
[INFO] Rule 0: org.codehaus.mojo.extraenforcer.dependencies.EnforceBytecodeVersion passed
[INFO]
[INFO] --- enforcer:3.6.2:enforce (enforce-jdk-version) @ benchmark ---
[INFO] Rule 0: org.apache.maven.enforcer.rules.version.RequireJavaVersion passed
[INFO]
[INFO] --- resources:3.5.0:resources (default-resources) @ benchmark ---
[INFO] Copying 18 resources from src\main\resources to target\classes
[INFO]
[INFO] --- compiler:3.15.0:compile (default-compile) @ benchmark ---
[INFO] Recompiling the module because of changed source code.
[INFO] Compiling 2766 source files with javac [forked debug release 8] to target\classes
[INFO]
[INFO] --- spotless:3.4.0:apply (spotless-apply) @ benchmark ---
[INFO] Index file does not exist. Fallback to an empty index
[INFO]
[INFO] --- resources:3.5.0:testResources (default-testResources) @ benchmark ---
[INFO] skip non existing resourceDirectory C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark\src\test\resources
[INFO]
[INFO] --- compiler:3.15.0:testCompile (default-testCompile) @ benchmark ---
[INFO] No sources to compile
[INFO]
[INFO] --- surefire:3.5.5:test (default-test) @ benchmark ---
[INFO] No tests to run.
[INFO]
[INFO] --- war:3.5.1:war (default-war) @ benchmark ---
[INFO] Packaging webapp
[INFO] Assembling webapp [benchmark] in [C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark\target\benchmark]
[INFO] Processing war project
[INFO] Copying webapp resources [C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark\src\main\webapp]
[INFO] Building war: C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark\target\benchmark.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  48.600 s
[INFO] Finished at: 2026-05-11T18:34:00+02:00
[INFO] ------------------------------------------------------------------------
PS C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark>
```

Slika 23: Uspešno prevajanje izvorne kode projekta OWASP Benchmark

Vir: Lasten vir

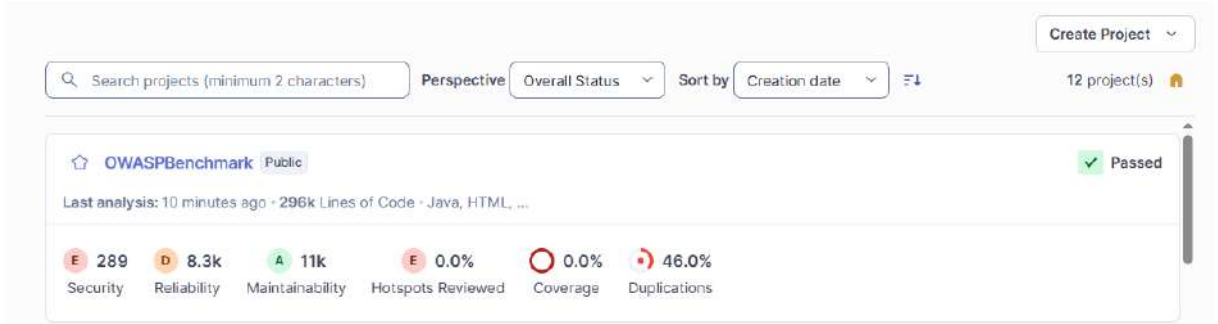
Po uspešnem prevajanju sem v isti mapi zagnal orodje sonar-scanner, ki je pregledalo projekt in rezultate analize poslalo na lokalni strežnik SonarQube.



Slika 24: Uspešen zagon ukaza sonar-scanner

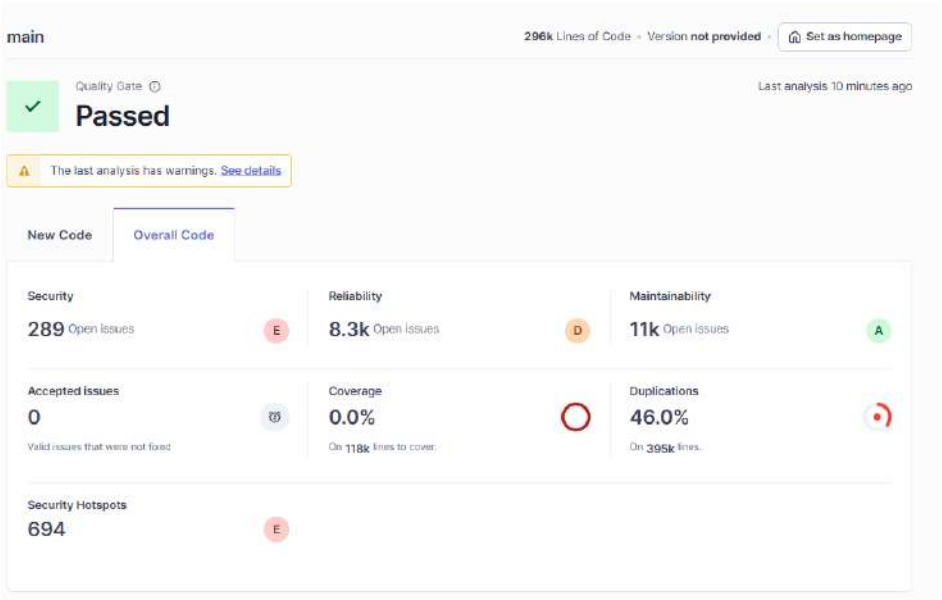
Vir: Lasten vir

Po zaključeni analizi sem se vrnil v spletni vmesnik SonarQube, kjer sem odprl projekt OWASP Benchmark in pregledal rezultate analize. Rezultati so bili prikazani v obliki pregleda projekta z osnovnimi kazalniki ter seznamom zaznanih ugotovitev. V tej fazi me ni zanimalo le, ali je analiza tehnično uspela, temveč tudi, kakšen obseg varnostnih in drugih opozoril je SonarQube vrnil nad standardiziranim testnim naborom.



Slika 25: Projekt OWASPBenchmark v SonarQube CE

Vir: Lasten vir

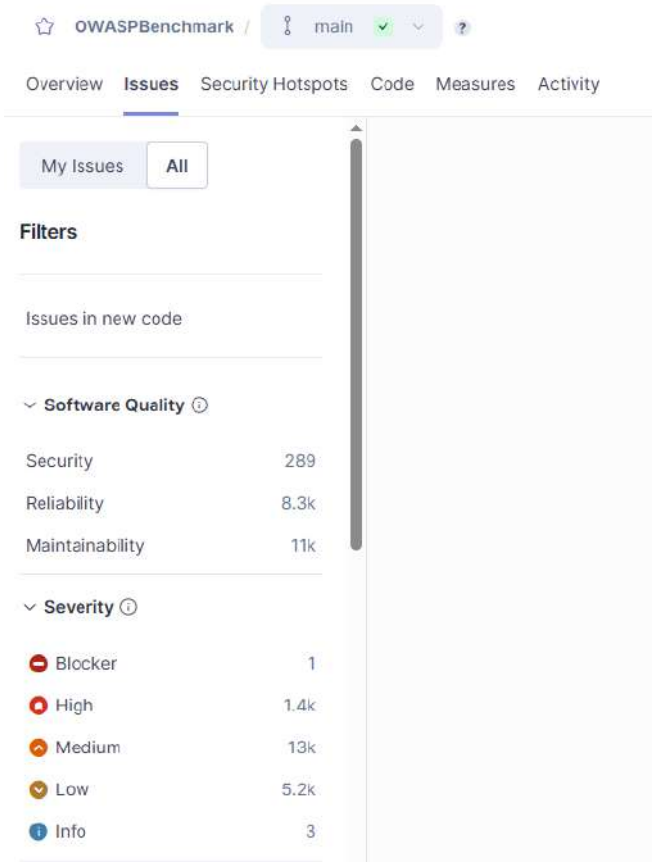


Slika 26: Osnovni pregled rezultatov analize projekta OWASP Benchmark v SonarQube CE

Vir: Lasten vir

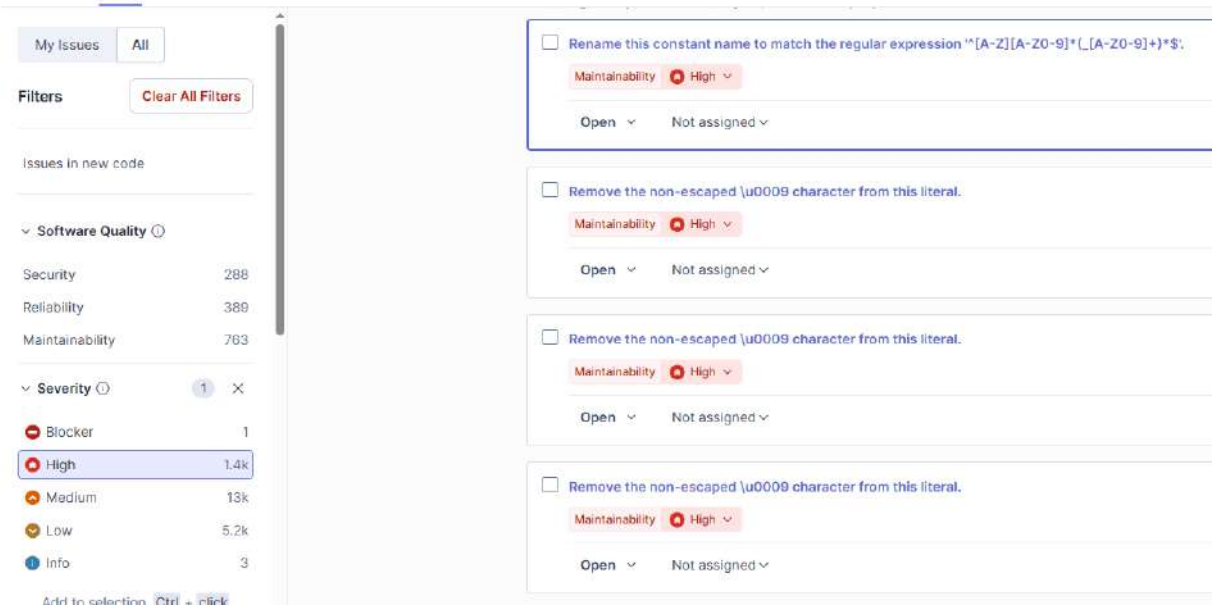
SonarQube CE je rezultate prikazal v obliki pregledne nadzorne plošče projekta. Ločeno so prikazani varnostni problemi (Security), težave zanesljivosti (Reliability), problemi vzdrževanja (Maintainability), varnostne točke pregleda (Security Hotspots), podvajanje kode (Duplications) in pokritost s testi (Coverage). Tak pregled omogoča hiter vpogled v splošno stanje analiziranega projekta, vendar hkrati pokaže, da SonarQube CE vrne zelo veliko število različnih opozoril, med katerimi niso vsa neposredno uporabna za obravnavo in kategorizacijo izbranih ranljivosti.

V zavihku Issues SonarQube CE zaznana opozorila razvrsti glede na kakovostno kategorijo in stopnjo resnosti. Tak prikaz omogoča hitrejši pregled nad velikim številom najdb, hkrati pa pokaže, da orodje vrne zelo širok nabor opozoril, med katerimi niso vsa neposredno relevantna za izbrane tipe ranljivosti.



Slika 27: Razvrstitev zaznanih opozoril v zavihku Issues

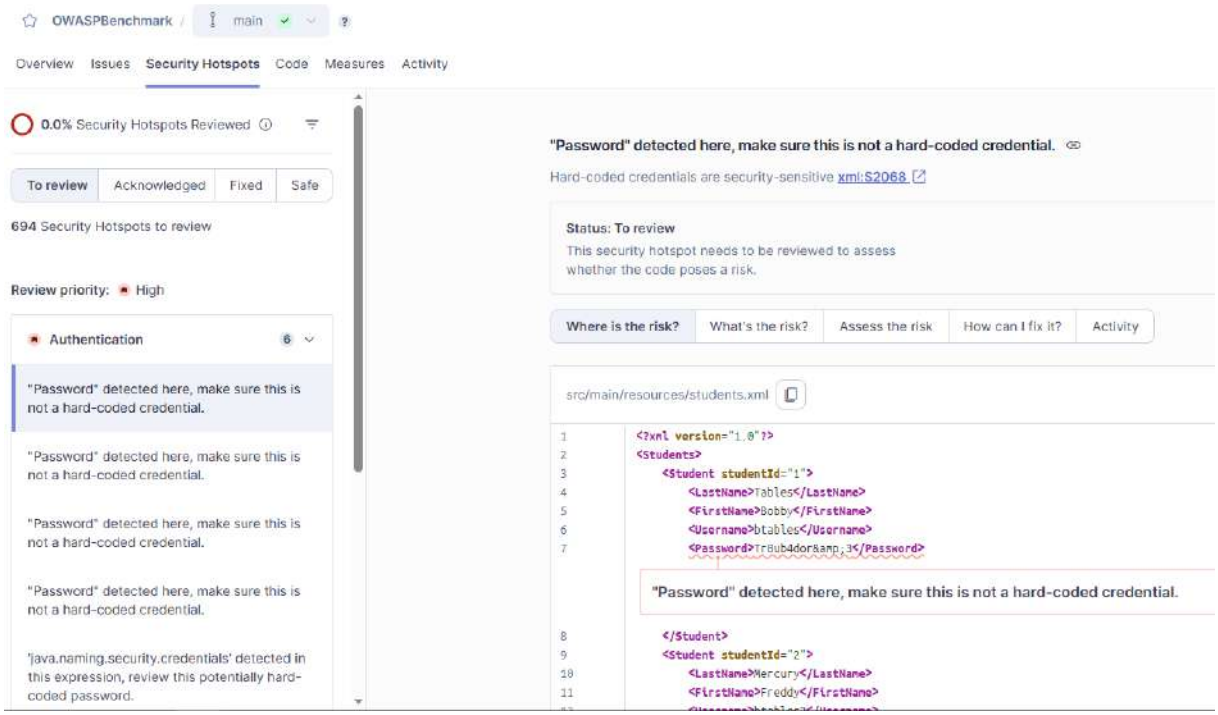
Vir: Lasten vir



Slika 28: Podrobnejši prikaz posameznega opozorila v zavihku Issues

Vir: Lasten vir

V zavihku Security Hotspots SonarQube CE prikaže mesta v kodi, ki po presoji orodja zahtevajo dodaten varnostni pregled. Takšne zaznave niso nujno potrjene ranljivosti, temveč opozorila, pri katerih mora uporabnik ročno presoditi, ali predstavljajo dejansko varnostno tveganje.



Slika 29: Primer zaznave v zavihku Security Hotspots

Vir: Lasten vir

S tem je bila analiza projekta OWASP Benchmark z orodjem SonarQube CE uspešno izvedena. SonarQube CE je rezultate prikazal v več različnih razdelkih, kar je omogočilo pregled nad obsegom zaznanih opozoril in pokazalo potrebo po dodatni selekciji relevantnih zaznav za nadaljnjo primerjavo z orodjem Semgrep CE.

6.1.3 Izvedba analize z orodjem Semgrep CE

Po zaključeni analizi z orodjem SonarQube CE sem nad istim projektom OWASP Benchmark izvedel še analizo z orodjem Semgrep CE. Za primerljivo primerjavo sem uporabil isti izvorni projekt. Ker Semgrep CE ne uporablja lokalnega strežnika ali spletnega uporabniškega vmesnika, se analiza izvaja neposredno iz ukazne vrstice. Pred zagonom analize sem zato najprej odprl korensko mapo projekta OWASP Benchmark v ukazni vrstici.

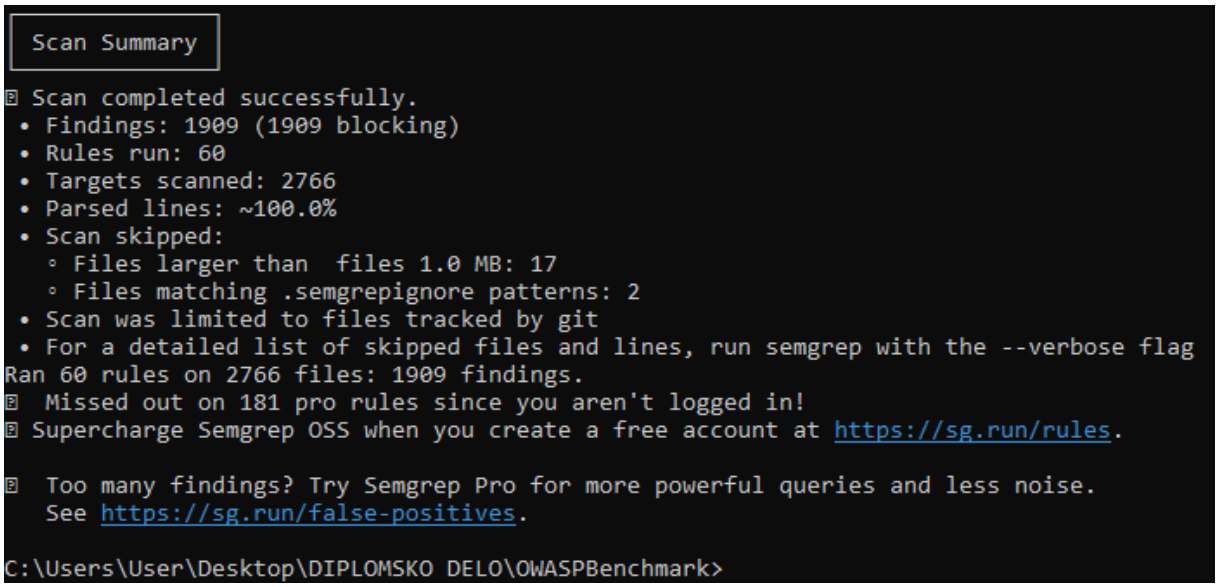
Ker v mojem sistemskem okolju ukaz semgrep ni bil neposredno dostopen prek sistemske poti PATH, sem pri analizi uporabil celotno pot do izvršljive datoteke semgrep.exe. Tako sem zagotovil pravilno izvajanje orodja brez dodatnega spreminjanja sistemskih nastavitev. Za analizo Java projekta sem uporabil javno dostopen nabor pravil p/java, ki vsebuje pravila za pregled kode v programskem jeziku Java.

```
C:\Users\User\Desktop\DIPLOMSKO DELO\OWASPBenchmark>"C:\Users\User\AppData\Local\Programs\Python\Python312
Scripts\semgrep.exe" scan --config=p/java .
```

Slika 30: Ukaz za zagon analize nad projektom

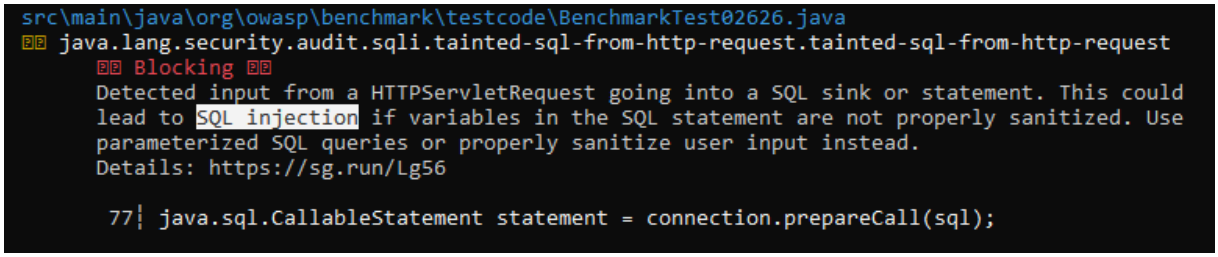
Vir: Lasten vir

Po zagonu je Semgrep CE pregledal projektne datoteke in rezultate izpisal neposredno v ukazni vrstici. Izpis je vseboval informacije o številu pregledanih datotek, številu uporabljenih pravil in številu zaznanih ugotovitev. Poleg skupnega povzetka rezultatov je Semgrep CE pri posameznih zaznavah izpisal tudi konkretno datoteko projekta, ime pravila, opis varnostnega problema in vrstico kode, na katero se opozorilo nanaša. Tak prikaz je bil pri praktični analizi posebej uporaben in je prikazal neposredno povezavo med zaznano ranljivostjo in posameznim testnim primerom projekta OWASP Benchmark.



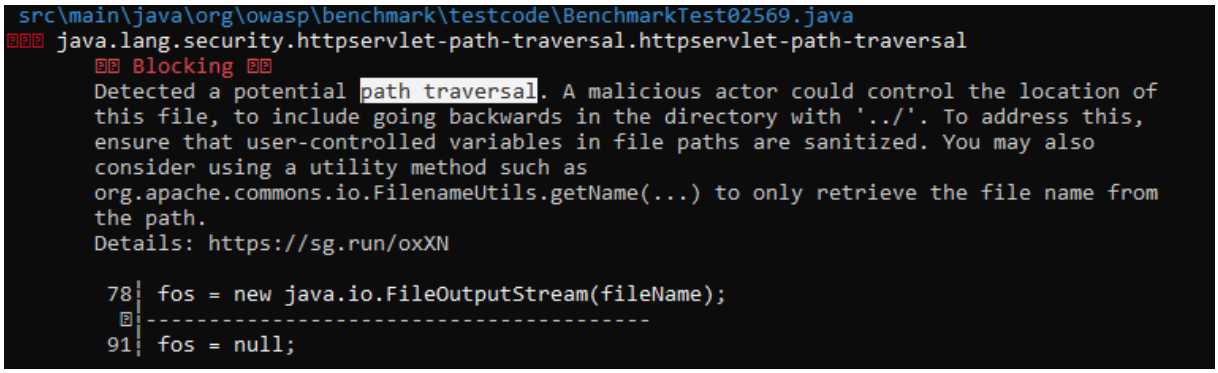
Slika 31: Rezultat analize projekta z orodjem Semgrep

Vir: Lasten vir



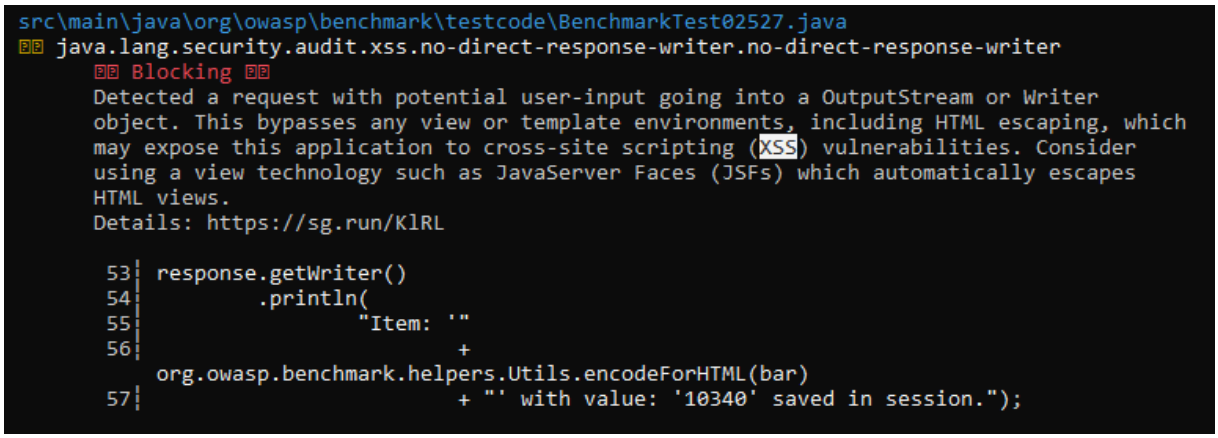
Slika 32: Primer zaznave SQL Injection v orodju Semgrep

Vir: Lasten vir



Slika 33: Primer zaznave Path Traversal v orodju Semgrep

Vir: Lasten vir



Slika 34: Primer zaznave XSS v orodju Semgrep

Vir: Lasten vir

Kot je razvidno iz prikazanih primerov, je Semgrep CE pri zaznavah jasno navedel, da gre za potencialne primere SQL Injection, Path Traversal ter XSS, in jih povezal s točno določenimi datotekami BenchmarkTestXXXX.java. To je olajšalo nadaljnjo izbiro manjšega nabora reprezentativnih testnih primerov za podrobnejšo primerjavo med orodjema.

Na ta način je bila analiza projekta OWASP Benchmark tudi z orodjem Semgrep CE zaključena. V primerjavi s SonarQube CE je bil postopek tehnično preprostejši, saj ni zahteval ločenega zagona strežnika, ustvarjanja projekta v spletnem vmesniku ali priprave dodatne konfiguracije povezave. V primerjavi s SonarQube CE je Semgrep CE pri izpisu rezultatov deloval bolj pregledno in neposredno, saj je pri posameznih zaznavah jasneje nakazal tip ranljivosti ter jo povezal s konkretno datoteko in vrstico kode. Zaradi takšnega prikaza je bila identifikacija relevantnih primerov preprostejša in za uporabnika tudi bolj razumljiva.

6.1.4 Način beleženja rezultatov za preverjanje H1 in H2

Za preverjanje hipotez H1 in H2 sem kot referenčno osnovo uporabil datoteko expectedresults-1.2.csv, ki je del projekta OWASP Benchmark. Ta datoteka za posamezne testne primere vsebuje ime testa, kategorijo ranljivosti, pripadajočo oznako CWE in informacijo o tem, ali gre za dejansko ranljiv ali varen primer. Na tej osnovi sem oblikoval enak vzorčni sklop testnih primerov za obe orodji.

V vzorec sem vključil skupaj 30 testov, in sicer po 10 testov za vsako izmed treh obravnavanih ranljivosti: SQL Injection, Path Traversal in XSS. Pri vsaki ranljivosti sem izbral 5 testnih

primerov, ki so bili po referenčnih podatkih označeni kot dejansko ranljivi (true), in 5 testnih primerov, ki so bili označeni kot varni (false).

Takšna razporeditev je omogočila, da sem pri obeh orodjih preverjal ne le uspešnost zaznavanja dejanskih ranljivosti, ampak tudi pojav lažno pozitivnih zaznav. Enak vzorčni sklop je bil uporabljen pri obeh orodjih, da sem zagotovil primerljive pogoje za nadaljnjo analizo. Za vsak testni primer sem nato preveril, kaj je posamezno orodje vrnilo za pričakovano vrsto ranljivosti.

V Tabela 4 je prikazan izbrani vzorčni sklop testnih primerov, ki je bil uporabljen za preverjanje hipotez H1 in H2, skupaj z rezultati obeh orodij pri posameznem testnem primeru.

Tabela 4: Rezultati analize vzorčnega sklopa z obema orodjema

Test	Ranljivost	CWE	Ground truth	SonarQube zaznava	Semgrep zaznava
BenchmarkTest00008	SQL Injection	89	True	0	1
BenchmarkTest00018	SQL Injection	89	True	0	1
BenchmarkTest00024	SQL Injection	89	True	0	1
BenchmarkTest00025	SQL Injection	89	True	0	1
BenchmarkTest00026	SQL Injection	89	True	0	1
BenchmarkTest00052	SQL Injection	89	False	0	0
BenchmarkTest00104	SQL Injection	89	False	0	1
BenchmarkTest00105	SQL Injection	89	False	0	1

BenchmarkTest00107	SQL Injection	89	False	0	0
BenchmarkTest00110	SQL Injection	89	False	0	0
BenchmarkTest00013	XSS	79	True	0	1
BenchmarkTest00014	XSS	79	True	0	1
BenchmarkTest00030	XSS	79	True	0	1
BenchmarkTest00036	XSS	79	True	0	1
BenchmarkTest00041	XSS	79	True	0	0
BenchmarkTest00147	XSS	79	False	0	0
BenchmarkTest00151	XSS	79	False	0	0
BenchmarkTest00275	XSS	79	False	0	0
BenchmarkTest00277	XSS	79	False	0	0
BenchmarkTest00278	XSS	79	False	0	0
BenchmarkTest00001	Path Traversal	22	True	0	1
BenchmarkTest00002	Path Traversal	22	True	0	1
BenchmarkTest00011	Path Traversal	22	True	0	1
BenchmarkTest00028	Path Traversal	22	True	0	1
BenchmarkTest00040	Path Traversal	22	True	0	1
BenchmarkTest00063	Path Traversal	22	False	0	1

BenchmarkTest00064	Path Traversal	22	False	0	1
BenchmarkTest00131	Path Traversal	22	False	0	1
BenchmarkTest00132	Path Traversal	22	False	0	1
BenchmarkTest00134	Path Traversal	22	False	0	0

Vir: Lasten vir

Iz rezultatov v Tabela 4 je razvidno, da SonarQube CE v izbranem vzorčnem sklopu ni vrnil nobene relevantne zaznave za obravnavane ranljivosti SQL Injection, XSS in Path Traversal. To pomeni, da pri nobenem izmed ranljivih testnih primerov ni zaznal pričakovane vrste ranljivosti. SonarQube CE je sicer pri analizi projekta OWASP Benchmark vrnil veliko število opozoril, vendar ta v obravnavanem vzorcu večinoma niso bila vsebinsko povezana z izbranimi ranljivostmi. Šlo je predvsem za priporočila glede vzdrževanja kode, slogovna priporočila in druga splošna opozorila.

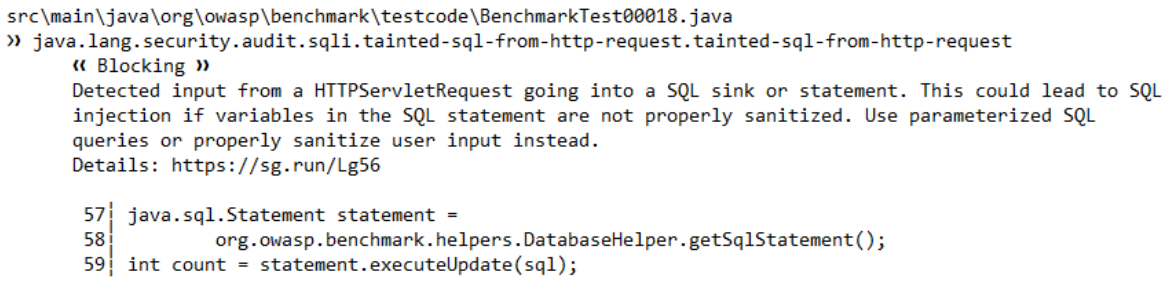
Semgrep CE je na drugi strani pri istem vzorčnem sklopu vrnil več relevantnih zaznav. Pri testnih primerih za SQL Injection in Path Traversal je zaznal vse izbrane ranljive primere, pri XSS pa večino izbranih ranljivih primerov. Hkrati se je pri nekaterih neranljivih testnih primerih pokazalo tudi nekaj lažno pozitivnih zaznav, predvsem pri SQL Injection in Path Traversal.

Primer zaznave obeh orodij nad testom BenchmarkTest00018.java:



Slika 35: Rezultat SonarQube CE nad testom BenchmarkTest00018.java

Vir: Lasten vir



Slika 36: Rezultat Semgrepa nad testom BenchmarkTest00018.java

Vir: Lasten vir

Na osnovi teh rezultatov sem za obe orodji izračunal še osnovne metrike uspešnosti zaznavanja, in sicer TP (true positive), FP (false positive), FN (false negative) in TN (true negative). Za vsako orodje sem izdelal tabelo z metrikami in s pomočjo teh podatkov izračunal tudi precision, recall in F1. Rezultati so prikazani v Tabela 5: Metrike uspešnosti zaznavanja s SonarQube CE.

Tabela 5: Metrike uspešnosti zaznavanja s SonarQube CE

SonarQube CE	Ranljivost	TP	FP	FN	TN
	SQL Injection	0	0	5	5
	XSS	0	0	5	5
	Path Traversal	0	0	5	5

Vir: Lasten vir

Precision, recall in F1 bomo izračunali po enačbah, omenjenih v poglavju 3.4. Vse tri metrike moramo izračunati za vsako ranljivost posebej.

Kot je razvidno v Tabela 6: Izračun metrik analize s SonarQube CE, smo pri izračunu metrike precision za vse tri ranljivosti dobili rezultat, pri katerem delimo z ničlo. Ker to v matematiki ni mogoče, smo zaradi lažjega razumevanja to vrednost upoštevali kot 0.

Tabela 6: Izračun metrik analize s SonarQube CE

$$Precision\ (SQL) = \frac{TP}{TP + FP} = \frac{0}{0 + 0} = ni\ definirano$$

$$Recall(SQL) = \frac{TP}{TP + FN} = \frac{0}{0 + 5} = 0$$

$$F1(SQL) = 2 \times \frac{0\ x\ 0}{0 + 0} = 0$$

$$Precision\ (XSS) = \frac{TP}{TP + FP} = \frac{0}{0 + 0} = 0$$

$$Recall(XSS) = \frac{TP}{TP + FN} = \frac{0}{0 + 5} = 0$$

$$F1(XSS) = 2 \times \frac{0\ x\ 0}{0 + 0} = 0$$

$$Precision\ (Path\ Traversal) = \frac{TP}{TP + FP} = \frac{0}{0 + 0} = 0$$

$$Recall(Path\ Traversal) = \frac{TP}{TP + FN} = \frac{0}{0 + 5} = 0$$

$$F1(Path\ Traversal) = 2 \times \frac{0\ x\ 0}{0 + 0} = 0$$

Vir: Lasten vir

Pri orodju SonarQube CE v izbranem vzorcu ni bilo relevantnih pozitivnih zaznav, zato vrednosti precision po standardni formuli ni bilo mogoče izračunati, saj bi to pomenilo deljenje z 0. V tabeli je ta primer označen kot nedefiniran. Ker SonarQube CE ni zaznal nobenega ranljivega primera, je bila vrednost recall enaka 0, posledično pa je bila za namen primerjave kot 0 obravnavana tudi vrednost F1. Izračun metrik ponovimo še za orodje Semgrep.

Tabela 7: Metrike uspešnosti zaznavanja s Semgrep

Semgrep	Ranljivost	TP	FP	FN	TN
	SQL Injection	5	2	0	3
	XSS	4	0	1	5
	Path Traversal	5	4	0	1

Vir: Lasten vir

Tabela 8: Izračun metrik analize s Semgrep CE

$$Precision\ (SQL) = \frac{TP}{TP + FP} = \frac{5}{5 + 2} = \frac{5}{7} = 0{,}71$$

$$Recall(SQL) = \frac{TP}{TP + FN} = \frac{5}{5 + 0} = \frac{5}{5} = 1$$

$$F1(SQL) = 2 \times \frac{0{,}71\ x\ 1}{0{,}71 + 1} = 2\ x \frac{0{,}71}{1{,}71} = 2\ x\ 0{,}41 = 0{,}82$$

$$Precision\ (XSS) = \frac{TP}{TP + FP} = \frac{4}{4 + 0} = \frac{4}{4} = 1$$

$$Recall(XSS) = \frac{TP}{TP + FN} = \frac{4}{4 + 1} = \frac{4}{5} = 0{,}8$$

$$F1(XSS) = 2 \times \frac{1\ x\ 0{,}8}{1 + 0{,}8} = 2\ x \frac{0{,}8}{1{,}8} = 2\ x\ 0{,}44 = 0{,}88$$

$$Precision\ (Path\ Traversal) = \frac{TP}{TP + FP} = \frac{5}{5 + 4} = \frac{5}{9} = 0{,}56$$

$$Recall(Path\ Traversal) = \frac{TP}{TP + FN} = \frac{5}{5 + 0} = \frac{5}{5} = 1$$

$$F1(Path\ Traversal) = 2 \times \frac{0{,}56\ x\ 1}{0{,}56 + 1} = 2\ x \frac{0{,}56}{1{,}56} = 2\ x\ 0{,}36 = 0{,}72$$

Vir: Lasten vir

Iz rezultatov v Tabela 8: Izračun metrik analize s Semgrep CE je razvidno, da je Semgrep CE najvišjo vrednost F1 dosegel pri ranljivosti XSS, kjer je zaznal večino ranljivih primerov in ni vrnil lažno pozitivnih zaznav. Pri SQL Injection je prav tako dosegel visoko vrednost F1, nekoliko nižjo pa pri Path Traversal, kjer se je pokazalo več lažno pozitivnih zaznav. Kljub temu je pri vseh treh obravnavanih ranljivostih dosegel bistveno boljše rezultate kot SonarQube CE.

Iz izračunanih metrik vidimo, da se uspešnost obeh orodij pri izbranem vzorčnem sklopu precej razlikuje. SonarQube CE v obravnavanem vzorcu ni vrnil relevantnih zaznav za izbrane ranljivosti, Semgrep CE pa je pri vseh treh kategorijah vrnil uporabne pozitivne zaznave in dosegel izračunljive vrednosti precision, recall in F1.

Na osnovi zbranih podatkov lahko potrdimo oziroma ovržemo hipotezi **H1** in **H2**.

Našo prvo hipotezo H1, ki pravi, da bo orodje Semgrep na enakem standardnem testnem sklopu doseglo višjo ali enako natančnost kot SonarQube CE, lahko **potrdimo**. Kot je razvidno iz Tabela 6 in Tabela 8, Semgrep CE pri vseh tipih ranljivosti doseže večjo natančnost kot SonarQube CE. S tem smo kvantitativno dokazali potrditev hipoteze.

Hipoteza H2, ki pravi, da bo orodje SonarQube CE generiralo večje število lažno pozitivnih zaznav na enakem vzorčnem sklopu, pa lahko dokončno **ovržemo**. To pokažeta Tabela 5 in Tabela 7, iz katerih je razvidno, da je na vseh tipih ranljivosti orodje SonarQube CE skupno zaznalo 0 lažnih pozitivnih rezultatov, orodje Semgrep CE pa skupaj 6 lažnih pozitivnih rezultatov. Tudi s tem smo kvantitativno dokazali ovržitev hipoteze H2.

Raziskava na projektu OWASP Benchmark je s tem zaključena, v nadaljevanju pa se osredotočamo na lastno Java aplikacijo.

6.2 Analiza lastne Java aplikacije

Po analizi na testnem naboru OWASP Benchmark sem pripravil še lastno manjšo Java aplikacijo. Ta del praktičnega dela je bil namenjen predvsem preverjanju hipoteze H4. Tu me ni več zanimalo samo, katero orodje bolje zaznava ranljivosti, ampak predvsem, ali vključitev boljšega orodja v razvojni tok dejansko pomaga zmanjšati število ranljivosti v novi kodi.

Za ta del raziskave sem izbral orodje Semgrep CE. Za to odločitev je bilo več razlogov. Pri dosedanjem delu se je izkazalo, da je bil Semgrep CE preprostejši za namestitev in uporabo. Prav tako je bolj neposredno prikazoval vrsto ranljivosti, datoteko in mesto v kodi. Na testih

OWASP Benchmark je dosegel tudi boljše rezultate kot SonarQube CE pri ranljivostih, ki sem jih preverjal. Posledično se je izbira orodja Semgrep CE izkazala za ustreznejšo.

Pri tem delu raziskave sem pripravil dve različici iste aplikacije. Prva različica predstavlja izhodiščno stanje brez uporabe SAST med razvojem. Druga različica pa predstavlja isto aplikacijo po tem, ko sem pri razvoju sproti uporabljal Semgrep CE, upošteval njegova opozorila in popravljal kodo. Na ta način sem lahko primerjal, koliko ranljivosti je ostalo v prvi in koliko v drugi različici.

6.2.1 Zasnova simuliranega razvojnega primera

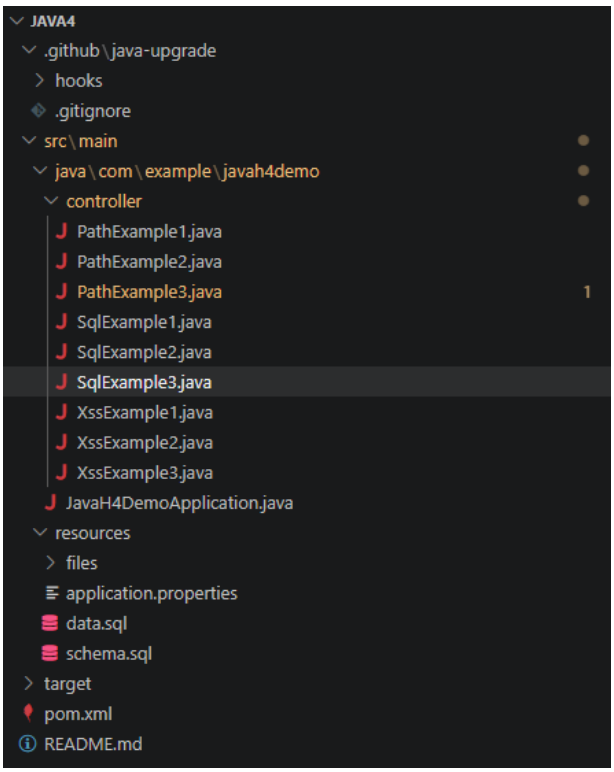
Za preverjanje H4 sem pripravil manjšo Java aplikacijo, ki je predstavljala kratek simuliran razvojni primer. Pri zasnovi tega dela sem izhajal tudi iz dejstva, da sem začetnik na področju varne programske opreme in uporabe orodij SAST. Ker se tudi naslov diplomskega dela osredotoča na primerjavo orodij za začetnike, se mi je zdelo smiselno, da je tudi praktični razvojni primer zasnovan na način, ki je začetniku blizu. To pomeni, da aplikacija ni bila pripravljena kot realen poslovni sistem, ampak kot manjši in pregleden projekt, v katerem so posamezni primeri dovolj jasni in razumljivi za praktično preverjanje.

V aplikacijo sem vključil tri tipe ranljivosti, ki sem jih obravnaval že v prejšnjem delu raziskave, in sicer XSS, SQL Injection in Path Traversal. Za vsako izmed teh ranljivosti sem pripravil po tri ločene razvojne primere, skupaj torej devet primerov. Vsi so bili del iste aplikacije, zaradi boljše preglednosti pa sem jih razdelil v ločene razrede. Takšna razdelitev je olajšala spremljanje posameznih primerov in kasnejšo primerjavo rezultatov.

Pri pripravi teh primerov si nisem pomagal zgolj s splošnim znanjem, temveč tudi z vzorci iz projekta OWASP Benchmark, ki sem ga uporabil v prejšnjem delu praktične analize. To se mi je zdelo smiselno, saj sem tam že videl, kakšne oblike kode orodje Semgrep CE dejansko zaznava. Zato sem posamezne primere v lastni aplikaciji pripravil tako, da so bili po svoji logiki podobni primerom iz projekta Benchmark, vendar še vedno dovolj preprosti in prilagojeni manjšemu lastnemu projektu.

Pomembno je tudi, da sem pri tem delu kot osnovno enoto štetja uporabil posamezen razvojni primer in ne posamezne najdbe orodja. Takšna odločitev je bila potrebna zato, ker lahko Semgrep CE pri isti kodi vrne več opozoril hkrati. Če bi štel vsako najdbo posebej, bi lahko isti ranljivi primer v rezultatih upošteval večkrat. Pri preverjanju H4 pa me je zanimal predvsem

odgovor na vprašanje, koliko ranljivih primerov je ostalo v kodi pred vključitvijo orodja v razvojni tok in po njej.



Slika 37: Struktura lastne Java aplikacije v okolju Visual Studio Code

Vir: Lasten vir

6.2.2 Izhodiščna različica aplikacije brez uporabe Semgrep CE

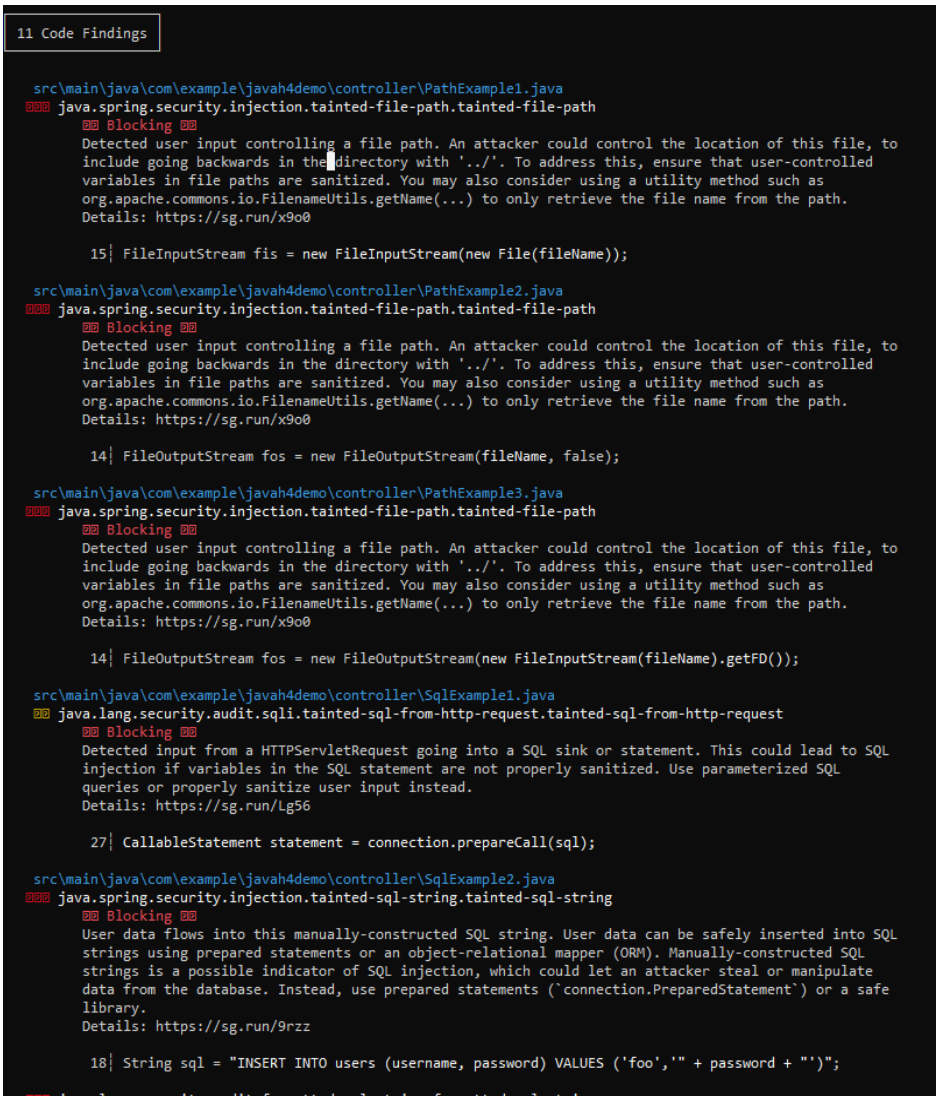
V prvem koraku sem pripravil izhodiščno različico aplikacije. Ta različica je predstavljala delo brez uporabe orodja SAST med razvojem. Pri pisanju kode v tej fazi nisem sproti preverjal opozoril orodja Semgrep CE, ampak sem najprej pripravil vse razvojne primere in jih analiziral šele na koncu.

Cilj tega koraka je bil vzpostaviti jasno začetno stanje. Zanimalo me je, koliko ranljivih primerov bo ostalo v kodi, če med razvojem ne uporabljam orodja za statično varnostno analizo. Takšna različica je bila zato namenoma pripravljena tako, da so v njej ranljivosti ostale prisotne.

V izhodiščni aplikaciji sem pripravil devet razvojnih primerov. Trije so bili z ranljivostjo XSS, trije s SQL Injection in trije s Path Traversal. Pri primerih XSS sem uporabniški vnos neposredno vključil v odgovor HTTP. Pri primerih SQL Injection sem uporabniške vrednosti

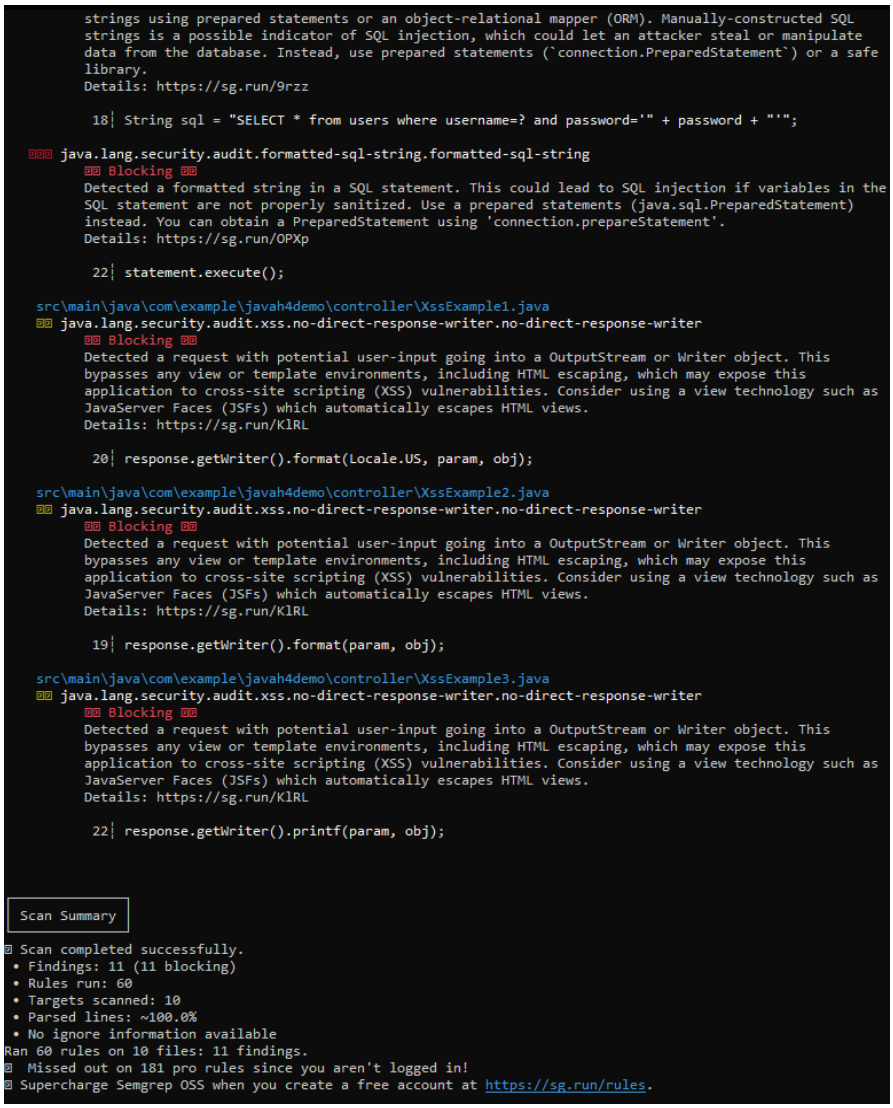
vključil v poizvedbe SQL na nevaren način. Pri primerih Path Traversal pa je uporabniški vnos vplival na delo z datotekami oziroma na oblikovanje poti do njih.

Ko je bila izhodiščna različica aplikacije pripravljena, sem nad celotnim projektom izvedel analizo z orodjem Semgrep CE.



Slika 38: Rezultati analize izhodiščne različice v Semgrep CE 1/2

Vir: Lasten vir



Slika 39: Rezultati analize izhodiščne različice v Semgrep CE 2/2

Vir: Lasten vir

Analiza je vrnila skupno 11 opozoril. To pa še ne pomeni 11 različnih ranljivih primerov. Pri dveh primerih SQL Injection je Semgrep CE za isti del kode vrnil po dve opozorili. Zato pri tem delu nisem štel posameznih najdb, ampak posamezne razvojne primere.

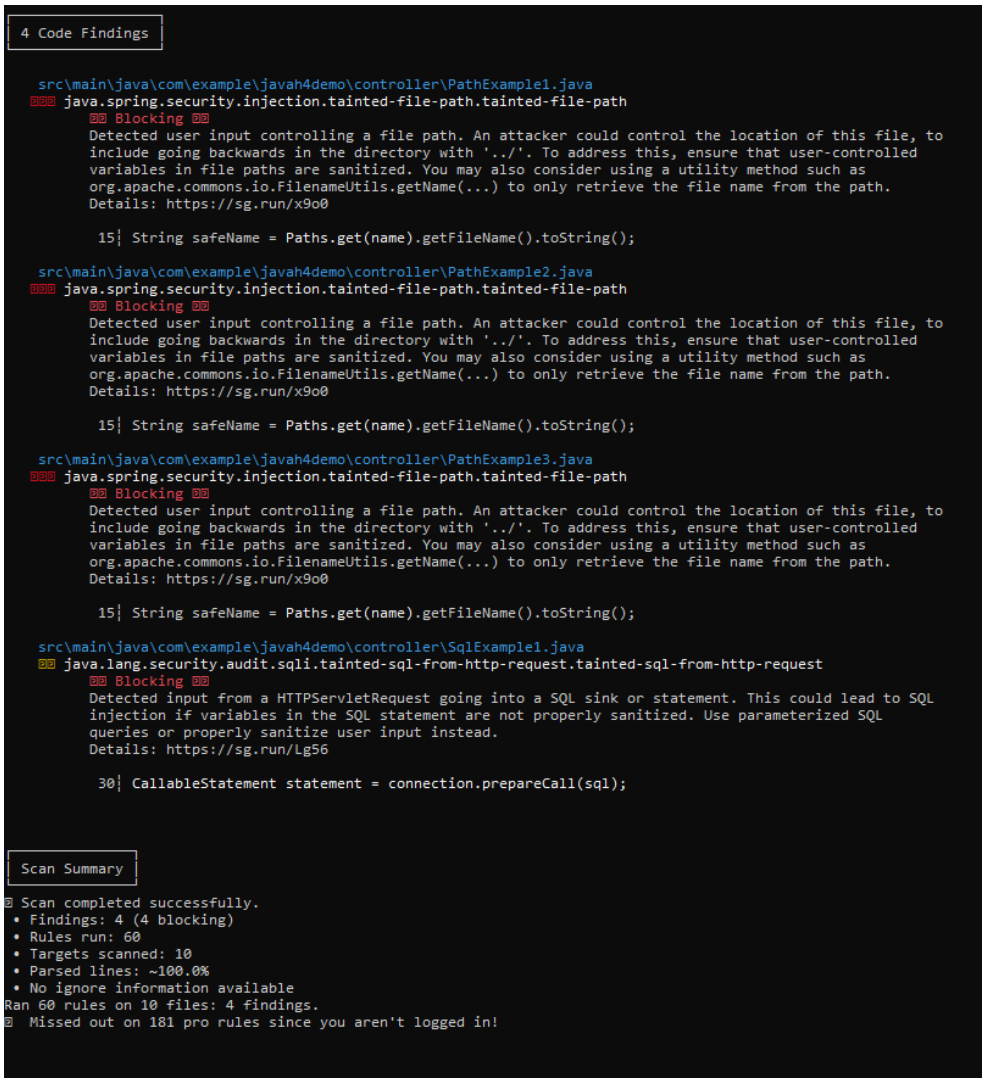
Ko sem rezultate pregledal na ta način, sem ugotovil, da je bilo v izhodiščni različici zaznanih vseh 9 od 9 pripravljenih razvojnih primerov. To pomeni, da je Semgrep CE v tej fazi zaznal vse tri primere XSS, vse tri primere SQL Injection in vse tri primere Path Traversal.

6.2.2 Različica aplikacije s sprotno uporabo Semgrep CE

Po pripravi izhodiščne različice sem pripravil še izboljšano različico iste aplikacije. Pri tej različici sem Semgrep CE uporabljal sproti med delom. Kodo sem po spremembah sproti preverjal z analizo v Semgrep CE in glede na izpis popravljal posamezne primere. Ker nimam veliko praktičnih izkušenj z varnim razvojem, sem pri popravljanju kode izhajal predvsem iz opozoril, ki jih je vrnilo orodje. Pomagal sem si tudi z osnovnimi priporočili iz opisov pravil, internetnimi programerskimi forumi in z logiko, ki sem jo spoznal pri prejšnjem delu raziskave na OWASP Benchmark. Na ta način nisem popravljal kode »na pamet«, želel pa sem se tudi sproti nekaj naučiti.

Pri izboljšani različici sem ohranil iste razvojne primere, spremenil pa sem način implementacije. Pri primerih XSS sem se izognil neposrednemu prikazu uporabniškega vnosa v odgovor. Pri primerih SQL Injection sem opustil sestavljanje poizvedb SQL iz nevarno združenih nizov in prešel na varnejši pristop. Pri primerih Path Traversal pa sem skušal bolj omejiti uporabniški vhod, da ta ne bi več tako neposredno vplival na pot do datoteke. Ideje za te popravke sem dobil predvsem iz pregleda testov iz projekta OWASP Benchmark, ki so bili označeni kot varni in neranljivi, ter s spletnih platform, kot sta Stack Overflow in podobne.

Ko je bila izboljšana različica pripravljena, sem nad projektom ponovno izvedel analizo z orodjem Semgrep CE.



Slika 40: Rezultati analize izboljšane različice s Semgrep CE

Vir: Lasten vir

Rezultati so pokazali, da je Semgrep CE pri izboljšani različici vrnil 4 relevantne zaznave. To pomeni, da so po sprotnem preverjanju in popravkih v kodi ostali zaznani 4 od 9 razvojnih primerov.

Ugotovimo lahko tudi, da so pri izboljšani različici vse tri zaznave za Path Traversal ostale prisotne. Torej spremembe pri teh primerih niso bile dovolj, da bi jih Semgrep CE prenehal obravnavati kot problematične. Kljub temu se je skupno število zaznanih ranljivih primerov v primerjavi z izhodiščno različico zmanjšalo. To kaže, da so bile spremembe pri nekaterih drugih primerih uspešnejše, medtem ko so primeri dela z datotekami tudi po omejitvi uporabniškega vnosa ostali bolj občutljiv del kode.

6.2.3 Primerjava rezultatov za preverjanje H4

Izhodiščna različica aplikacije je vsebovala 9 pripravljenih primerov. Pri vseh devetih je Semgrep CE vrnil relevantno zaznavo. To pomeni, da je bilo v izhodiščni različici zaznanih 9 od 9 ranljivih primerov. Pri izboljšani različici aplikacije je Semgrep CE vrnil relevantne zaznave še pri 4 od 9 primerov. Na tej osnovi sem lahko izračunal, za koliko se je število zaznanih ranljivih primerov zmanjšalo po vključitvi Semgrep CE v simulirani razvojni tok.

Za izračun zmanjšanja sem uporabil naslednjo enačbo:

Tabela 9: Izračun zmanjšanja napak v razvojnem procesu pri sprotni uporabi Semgrep CE

$$izhodiščna\ vrednost = število\ zaznav\ v\ izhodiščni\ različici\ aplikacije$$

$$izboljšana\ vrednost = število\ zaznav\ v\ izboljšani\ različici\ aplikacije$$

$$zmanjšanje\ zaznav = \frac{izhodiščna\ vrednost - izboljšana\ vrednost}{izhodiščna\ vrednost} x\ 100$$

$$= \frac{9-4}{9}\ x\ 100 = \frac{5}{9} x\ 100 = 0{,}556\ x\ 100 = 55{,}6\%$$

Vir: Lasten vir

Ugotovil sem, da je bilo v izboljšani različici aplikacije zaznanih za 55,6 % manj ranljivih razvojnih primerov kot v izhodiščni različici brez uporabe SAST.

Torej je sprotna uporaba orodja Semgrep CE med razvojem vplivala na manjše število ranljivosti v končni kodi. Pri tem se sicer ni pokazalo, da bi bili uspešno odpravljeni vsi primeri. Posebej pri Path Traversal so vsa tri opozorila ostala prisotna tudi po spremembah kode. Kljub temu je bila razlika med obema različicama še vedno dovolj velika, da je bilo zmanjšanje števila zaznanih ranljivih primerov jasno opazno.

Ta del raziskave je bil pripravljen kot manjši in nadzorovan simuliran primer, zato ga ni mogoče razumeti kot splošno oceno vseh možnih razvojnih situacij. Je pa dovolj dober pokazatelj, da lahko vključitev izbranega orodja v razvojni tok tudi pri začetniku prispeva k manjšemu številu varnostnih pomanjkljivosti v novi kodi.

Na osnovi zbranih podatkov lahko zaključim presojo zadnje hipoteze H4. Kot je razvidno iz Tabela 9, se je ob sprotni uporabi statične analize kode v kratkem simuliranem sprintu število

zaznanih ranljivih primerov zmanjšalo za 55,6 %. Ker je to več kot 30 %, kot je navedeno v H4, lahko to hipotezo **potrdim**.

7 SKLEP

V diplomskem delu sem si zadal cilj primerjati orodji SonarQube CE in Semgrep CE ter ugotoviti, katero izmed njiju je primernejše za začetnika pri odkrivanju varnostnih ranljivosti v kodi. Zanimala me je uspešnost orodij pri zaznavanju izbranih ranljivosti, kako pregledni in uporabni so njuni rezultati ter kako zahtevna je njuna vključitev v praktično delo.

Na osnovi teoretičnega in praktičnega dela sem ugotovil, da se med orodjema pojavljajo precejšnje razlike. Pri analizi testnih primerov OWASP Benchmark se je pokazalo, da je Semgrep CE pri izbranih ranljivostih vračal bolj neposredne in uporabnejše zaznave kot SonarQube CE. SonarQube CE je sicer vrnil veliko število opozoril, vendar ta pri mojem vzorcu večinoma niso bila vsebinsko povezana z ranljivostmi SQL Injection, XSS in Path Traversal, ki so bile predmet raziskave. Semgrep CE je bil pri teh ranljivostih jasnejši, preglednejši in tudi uspešnejši.

Med pripravo praktičnega dela sem pridobil tudi več konkretnega znanja. Bolje sem razumel delovanje statične analize izvorne kode, razliko med relevantnimi zaznavami in lažno pozitivnimi rezultati ter uporabo metrik precision, recall in F1. Poleg tega sem pridobil boljši vpogled v praktično uporabo obeh orodij, njuno namestitev, pripravo okolja in vključitev v razvojni tok. Pomembna pridobitev je bilo tudi boljše razumevanje tega, kako lahko začetnik postopno uporablja orodje SAST kot pomoč pri pisanju in popravljanju kode.

H1: Semgrep bo na standardnih testnih primerih dosegel enako ali višjo natančnost (F1) kot SonarQube CE za izbrane CWE (SQLi, XSS, Path Traversal) pri istem jeziku.

Hipoteza H1 je bila **potrjena**, kot smo razložili v poglavju 6.1.4. Pri izbranem vzorčnem sklopu testov je Semgrep pri vseh treh obravnavanih ranljivostih dosegel višje vrednosti F1 kot SonarQube CE.

H2: SonarQube CE bo generiral več lažno pozitivnih rezultatov kot Semgrep CE na enakem vzorčnem sklopu.

Hipoteza H2 je bila **ovržena**, kot smo pojasnili v poglavju 6.1.4. V izbranem vzorcu je več lažno pozitivnih zaznav vrnil Semgrep, medtem ko SonarQube CE pri obravnavanih ranljivostih ni vračal relevantnih zaznav.

H3: Začetnik bo hitreje namestil in povezal Semgrep (CLI/IDE/CI) kot SonarQube CE ter hitreje razumel razlago napak.

Hipoteza H3 je bila **potrjena** v poglavju 5.2.3. Pri praktičnem delu se je pokazalo, da je bil Semgrep CE preprostejši za namestitev, hitrejši za prvi zagon in bolj razumljiv pri razlagi zaznav kot SonarQube CE. Omeniti moramo tudi, da argumentacija te hipoteze temelji na subjektivni oceni. Za natančnejšo presojo te hipoteze bi bila potrebna večja skupina uporabnikov, ki bi vsak na svoj način opisali svoje občutke in izkušnje pri konfiguraciji, zagonu ter uporabi teh orodij. Meni osebno je bilo orodje Semgrep CE veliko lažje, hitrejše za razumevanje in uporabo kot SonarQube CE.

H4: Vključitev izbranega orodja v razvojni tok zmanjša število vnesenih ranljivosti v novi kodi za ≥ 30 % v primerjavi z izhodiščem brez SAST v kratkem simuliranem sprintu.

Hipoteza H4 je bila **potrjena** v poglavju 6.2.3. Pri lastni Java aplikaciji se je po vključitvi Semgrep CE v simuliran razvojni tok število zaznanih ranljivih primerov zmanjšalo za 55,6 %.

Pri izdelavi diplomskega dela sem ugotovil, kako hitro lahko že v preprosti kodi nastanejo varnostne ranljivosti in kako pomembno je, da jih odkrijemo čim prej. Prav zato se je med raziskavo pokazal velik pomen statične analize izvorne kode. Orodja SAST razvijalca sicer ne morejo nadomestiti, lahko pa mu pomagajo pri hitrejšem prepoznavanju tveganj in pri varnejšem razvoju kode.

Ob tem se mi zdi pomembno poudariti tudi, da so rezultati moje raziskave vezani na konkretno izbrano okolje, uporabljene različice orodij in način izvedbe primerjave. Posebej je treba upoštevati, da sem v raziskavi uporabljal brezplačni različici obeh orodij. Menim, da bi bili rezultati lahko drugačni, če bi bila v primerjavo vključena tudi plačljiva različica enega ali obeh orodij. To še posebej velja za SonarQube, saj se je pri moji raziskavi pokazalo, da brezplačna različica pri izbranih ranljivostih ni vrnila relevantnih zaznav. Pri naprednejših oziroma plačljivih različicah bi bil rezultat lahko drugačen, zato teh ugotovitev ni mogoče nekritično posplošiti na celoten ekosistem obeh orodij.

Pomembna omejitev raziskave je tudi to, da je bil del primerjave, povezan s hipotezo H3, precej subjektivne narave. Pri tej hipotezi sem ocenjeval predvsem zahtevnost namestitve, priprave okolja in razumljivost razlage zaznav z vidika začetnika. Ker sem to izkušnjo opisoval na

osnovi lastnega dela, je mogoče, da bi kdo drug ob enaki raziskavi prišel do nekoliko drugačnih ugotovitev. Zanimivo bi bilo preveriti, kako bi isti primerjalni postopek ocenili drugi začetniki ali razvijalci z drugačnim predznanjem in izkušnjami.

Za nadaljnje raziskovanje bi bilo smiselno primerjavo razširiti še na druga orodja, druge programske jezike ali bolj kompleksne projekte. Smiselna nadgradnja tega diplomskega dela bi bila tudi primerjava plačljivih različic orodij, saj bi tak pristop omogočil širši vpogled v to, koliko dodatne funkcionalnosti dejansko vplivajo na kakovost zaznavanja in uporabniško izkušnjo. Upam, da se bo to v prihodnosti tudi zgodilo, saj bi mi bilo osebno to zelo zanimivo prebrati. Zanimivo bi bilo tudi preveriti, kako se rezultati razlikujejo pri bolj izkušenih razvijalcih in v naprednejših razvojnih okoljih. Po mojem mnenju bi bilo treba spodbujati tudi zgodnejšo uporabo orodij SAST pri učenju programiranja in v razvojni praksi, saj lahko že osnovna uporaba takšnih orodij pripomore k boljši varnosti kode.

8 LITERATURA IN VIRI

Algarni, A. M. (2022). The Historical Relationship between the Software Vulnerability Lifecycle and Vulnerability Markets: Security and Economic Risks. *Computers*, 137. doi:https://doi.org/10.3390/computers11090137

Dencheva, L. (2022). *Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools.* Dublin: National College of Ireland.

Ferreira Franco, E. (2020). *A Dynamical Evaluation Framework for Technical Debt Management in Software Maintenance Process.* São Paulo: Universidade de São Paulo.

Hovemeyer, D., & Pugh, W. (Oktober 2004). Finding Bugs Is Easy. *ACM SIGPLAN Notices*, 92–106.

Humayun, M., Jhanhji, N., Almufareh, M., & Khalil, M. (2022). Security Threat and Vulnerability Assessment and Measurement in Secure Software Development. *Computers, Materials & Continua*, 3009–3025. doi:https://doi.org/10.32604/cmc.2022.019289

Khan, M. E., & Khan, F. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications*, 12–15. Pridobljeno iz https://pdfs.semanticscholar.org/fe7a/4d40d8b17b1d143197840cc08c3a486cb618.pdf

Kuszczyński, K., & Walkowski, M. (2023). Comparative Analysis of Open-Source Tools for Conducting Static Code Analysis. *Sensors*. doi:https://doi.org/10.3390/s23187978

National Institute of Standards and Technology. (2020). *Security and Privacy Controls for Information Systems and Organizations.* Gaithersburg, MD: National Institute of Standards and Technology. doi:https://doi.org/10.6028/NIST.SP.800-53r5

OWASP. (2026). *BenchmarkJava*. Pridobljeno Marec 2026 iz GitHub: https://github.com/OWASP-Benchmark/BenchmarkJava

OWASP Foundation. (2021). *OWASP Top 10 – 2021: The Ten Most Critical Web Application Security Risks*. Pridobljeno Februar 2026 iz OWASP: https://owasp.org/Top10/2021/

Raschka, S. (21. Oktober 2014). *An Overview of General Performance Metrics of Binary Classifier Systems.* Pridobljeno iz arXiv: https://arxiv.org/abs/1410.5330

Semgrep. (2025). *CLI reference.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/cli-reference

Semgrep. (2025). *Generic pattern matching.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/writing-rules/generic-pattern-matching

Semgrep. (2025). *Semgrep Community Edition in CI.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/deployment/oss-deployment

Semgrep. (2025). *Semgrep Secrets overview.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/semgrep-secrets/conceptual-overview

Semgrep. (2026). *Docs home.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs

Semgrep. (2026). *Get started with Semgrep Community Edition.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/getting-started/quickstart-ce

Semgrep. (2026). *Introduction to Semgrep.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/introduction

Semgrep. (2026). *Run rules.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/running-rules

Semgrep. (2026). *Supported languages.* Pridobljeno Februar 2026 iz Semgrep Docs: https://semgrep.dev/docs/supported-languages

Semgrep. (Semgrep Docs). Pridobljeno Februar 2026 iz Semgrep Secrets overview: https://semgrep.dev/docs/semgrep-secrets/conceptual-overview

Sharma, S. (2025). Strategies for Integrating Security into the Software Development Lifecycle. *The American Journal of Engineering and Technology*, 151–156. doi:https://doi.org/10.37547/tajet/Volume07Issue04-20

Sinan, M., Shahin, M., & Gondal, I. (2025). Integrating Security Controls in DevSecOps: Challenges, Solutions, and Future Research Directions. *Journal of Software: Evolution and Process*. doi:https://doi.org/10.1002/smr.70029

SonarSource. (2026). *Analysis overview | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation:

https://docs.sonarsource.com/sonarqube-community-build/analyzing-source-code/analysis-overview

SonarSource. (2026). *Glossary | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/glossary

SonarSource. (2026). *Homepage | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build

SonarSource. (2026). *Introduction | SonarQube Server | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-server/advanced-security/introduction

SonarSource. (2026). *Managing quality profiles | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/quality-standards-administration/managing-quality-profiles

SonarSource. (2026). *Quality gates | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/quality-standards-administration/managing-quality-gates

SonarSource. (2026). *Rules | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/user-guide/rules/overview

SonarSource. (2026). *SonarQube rules | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/quality-standards-administration/managing-rules/rules

SonarSource. (2026). *Supported languages | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/analyzing-source-code/languages/overview

SonarSource. (2026). *Understanding quality profiles | SonarQube Community Build | Sonar Documentation.* Pridobljeno Februar 2026 iz Sonar Documentation: https://docs.sonarsource.com/sonarqube-community-build/quality-standards-administration/managing-quality-profiles/understanding-quality-profiles

The MITRE Corporation. (brez datuma). *Common Weakness Enumeration (CWE)*. Pridobljeno februar 2026 iz https://cwe.mitre.org/

Thomson, P. (julij/avgust 2021). Static Analysis: An Introduction. *ACM Queue*. Pridobljeno Marec 2026 iz https://queue.acm.org/detail.cfm?id=3487021

Yang, S., & Berdine, G. (2024). Confusion matrix. *The Southwest Respiratory and Critical Care Chronicles*, 75-79. doi:https://doi.org/10.12746/swrccc.v12i53.1391

Zgubič, M. (2015). *Statični analizator za preverjanje kvalitete programske kode.* Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko. Maribor: Univerza v Mariboru.

Zuva, K., & Zuva, T. (2012). Evaluation of Information Retrieval Systems. *International Journal of Computer Science & Information Technology*, 35–43. doi:10.5121/ijcsit.2012.4304

9 PRILOGE

- Uporabljena programska koda, oddana v nabiralniku v spletni učilnici