TARTU ÜLIKOOL

Arvutiteaduse instituut

Informaatika õppekava

Risto Voor

Kolmandate osapoolte teekide kontrollimise tööriistade analüüs ja täiustamine

Bakalaureusetöö (9 EAP)

Juhendaja: Kristiina Rahkema, PhD

Tartu 2024

Kolmandate osapoolte teekide kontrollimise tööriistade analüüs ja täiustamine

Lühikokkuvõte:

Käesolevas bakalaureusetöös analüüsiti erinevaid kolmandate osapoolte teekide kontrollimise tööriistu, koliti rakendus SwiftDependencyChecker programmeerimiskeelest Swift keelde Java ja analüüsiti erinevaid võimalusi selle täiustamiseks, ning realiseeriti üks nendest. Töös tutvustatakse lugejale erinevaid teemaga seotud termineid, tähtsamaid andmebaase ja tööriistu. Rakenduse kolimise kohta tuuakse välja selle põhjused ja tutvustatakse mõlemat keelt. Tööriista täiustamisel tutvustatakse kahte erinevat viisi, kuidas plaaniti seda teha ja miks realiseerus praegune lahendus.

Võtmesõnad: programmeerimine, turvalisus, kolmandate osapoolte teegid, Java, Swift

CERCS: P175 informaatika

Third-party libraries security tools analysis and improvements

Abstract:

In this bachelor's thesis, various third-party library checking tools were analyzed, the SwiftDependencyChecker application was moved from the Swift programming language to Java, and various ways to improve it were analyzed, and one of them was realized. The work introduces the reader to various terms related to the topic, the most important databases and tools. The reasons for moving the app are explained and both languages are introduced. Improving the tool will present two different ways of how it was planned to be done and why the current solution was realized.

Keywords: programming, security, dependencies, Java, Swift

CERCS: P175 Informatics

Sisukord

Sissejuhatus

Kolmandate osapoolte teegid on laialdaselt kasutatud erinevates projektides. Need võimaldavad programmeerijatel kasutada koodi, mis on juba testitud. Sellega saavad programmeerijad aega kokku hoida, kuna ei pea välja mõtlema lahendusi probleemidele, millel keegi on juba lahenduse leidnud [1]. Selle tulemus on kiirem ja lihtsam arendusprotsess.

Programmeerijal on kohustus kirjutada tarkvara, millega ei kahjusta teisi. Üks osa sellest on tagada programmi turvalisus. Osa sellest on programmeerijate enda kirjutatud kood, aga kasutades kolmandate osapoolte lahendusi tuleb vastutada ka sealt ilmnevate vigade puhul.

Selleks on loodud mitmeid avalikke andmebaase, milles on kirjeldatud haavatavustega kolmandate osapoolte teeke ja nende haavatavusi. On tööriistu, mis kontrollivad nende andmebaaside abil kasutajate projektides kasutatavate teekide versioone, sobitades neid olemasolevate ebaturvaliste teekide ja versioonidega. Nendega antakse kasutajatele teada kui nad peaksid oma projekte uuendama ja kui kriitiline see on, et vältida haavatavusi.

Hetkel üks selliseid tööriistu on SwiftDependencyChecker (siia joonealune viide panna). See kontrollib iOS-i ja macOS-i arenduses kasutuses olevate pakettidehaldurite Swift Package Manager, CocoaPods ja Carthage olevate teekide turvalisust. Programm on kirjutatud programmeerimiskeeles Swift ja mõeldud kasutamiseks eelkõige koos integreeritud programmeerimiskeskkonna Xcode-ga. Kuna nii Swift kui Xcode on mõeldud peamiselt iOS ja macOS programmide arendamiseks, siis SwiftDependencyChecker-i kasutamine teistel, eelkõige Windowsi ja Linux masinates, on raskendatud.

Üks lahendusi selleks on programm kolida programmeerimiskeelest Swift, laialdasemalt toetatud keelde Java. See võimaldab programmi kasutada rohkemate seadmete peal. Isegi kui SwiftDependencyChecker-i peamiseks kasutuskeskkonnaks jääb macOS keskkond, siis on situatsioone, kus ei ole alati võimalik macOS-i keskkonda kasutada.

Töö on jaotatud kolmeks peatükiks. Esimene annab taustainformatsiooni kolmandate osapoolte teekide kohta, nende halduritest, turvavigadest ja arendamisel kasutatud programmeerimiskeeltest. Teises peatükis analüüsitakse olemasolevaid kolmandate osapoolte teekide turvavigasid tuvastavate programme, antakse ülevaade kolimisprotsessist ja

analüüsitakse erinevaid võimalusi edasiarenduseks. Kolmandas tulemuste peatükis antakse ülevaade analüüsi tulemustest ja järeldustest, programmi kolimise edukusest ja juurdearenduse realiseerumisest.

1. Taust

1.1 Kolmandate osapoolte teegid

Kolmandate osapoolte teegid *(ingl third-party libraries*) [2] on eelnevalt loodud tarkvara komponendid või koodikogumid, mida arendajad saavad integreerida oma enda tarkvara projektidesse. Need teegid on väljastpoolt projekti loodud ja neid võivad arendada erinevad organisatsioonid või isikud.

Kolmandate osapoolte teegid võivad pakkuda erinevaid funktsioone ja teenuseid, nagu andmetöötlus, graafika renderdamine, võrguühendus, andmebaaside haldamine jne. Arendajad kasutavad neid teeke sageli selle asemel, et luua kõik vajalik tarkvara nullist, säästes sellega aega ja vaeva. Samuti on nende kasutamine enamikel juhtudel turvalisem, kuna kui tegemist on avatud lähtekoodiga, siis saavad kõik kasutajad pakkuda lahendusi, kuidas teeki paremaks teha ja ennetada potensiaalseid haavatavusi.

1.2 Paketihaldurid

Kolmandate osapoolte teekide pakettide haldurid on tööriistad, mis aitavad arendajatel hallata ja installida oma projektides kasutatavaid kolmandate osapoolte teeke. Need paketihaldurid lihtsustavad teekide allalaadimist, paigaldamist ja värskendamist [3]. Populaarsemad nendest on Gradle, Maven, npm, Nuget.

Swift-ile on kolm erinevat paketihaldurit: CocoaPods[1], Carthage[2], Swift PM[3]. Kõige populaarsem nendest CocoaPods, omades enim erinevaid pakette. Kõikide nende paketihaldurite pakettide turvalisust võimaldab kontrollida SwiftDependencyChecker.

1.3 Turvariskid

Kuigi kolmandate osapoolte teegid võivad olla kasulikud, on oluline jälgida nende turvalisust ja ajakohasust. Vananenud teegid võivad sisaldada turvaauke ja neid ei pruugita enam toetada, mis võib põhjustada probleeme tarkvaraprojekti turvalisuses ja stabiilsuses [4]. Seetõttu on

[1] https://cocoapods.org/
[2] https://github.com/Carthage/Carthage
[3] https://www.swift.org/documentation/package-manager/

soovitatav regulaarselt uuendada kasutatavaid kolmandate osapoolte teeke ja jälgida, et nendes ei oleks haavatavusi.

1.3.1 Turvariskide andmebaasid

On olemas mitmeid aktiivselt uuendatavaid andmebaase, kust on võimalik saada informatsiooni haavatavuste kohta, tuntumad nendest:

- **National Vulnerability Database (NVD)** [5] on USA riiklik andmebaas, mida haldab National Institute of Standards and Technology (NIST). NVD kogub, säilitab ja levitab teavet turvariskide, haavatavuste ja nendega seotud andmete kohta. Andmebaas sisaldab teavet erinevate tarkvarasüsteemide ja -rakenduste turvariskide kohta ning aitab organisatsioonidel ja turvaeksperdidel mõista, hinnata ja vähendada nende süsteemide turvariske.
- **GitHub Security Advisories** [6] on osa GitHubi platvormist, kus hoitakse ja hallatakse teavet turvaprobleemide kohta, millest on teatatud GitHubi projektides. See andmebaas võimaldab kasutajatel jälgida, teavitada ja lahendada turvahaavatavusi, mis on seotud nende tarkvaraprojektidega. GitHub Security Advisories andmebaas on osa GitHubi püüdlusest suurendada tarkvaraprojektide turvalisust ja edendada koostööd turvariskide haldamisel. See on oluline avatud lähtekoodiga projektide jaoks, kus lai kasutajaskond võib aidata tuvastada ja lahendada turvaprobleeme.
- **Snyk Vulnerability Database** [7] on ettevõte Snyk andmebaas, mis pakub lahendusi ja tööriistu tarkvara turvalisuse ja halduse parandamiseks.

1.3.2 Common Vulnerabilities and Exposures (CVE)

Common Vulnerabilities and Exposures (CVE) [8] on standardiseeritud süsteem haavatavuste ja ohustatud alade identifitseerimiseks infotehnoloogia ja küberturbe valdkonnas. CVE süsteemi eesmärk on anda unikaalsed identifikaatorid (CVE-ID-d) igale teadaolevale haavatavusele. See võimaldab jagada ühtset keelt, kui rääkida konkreetsetest turvariskidest, olenemata sellest, kas töötatakse tootearenduses või haavatavuste hindamisel.

CVE Details on veebipõhine platvorm, mis kogub, analüüsib ja esitab infot haavatavuste kohta, mis on seotud CVE-ID-dega [9]. See sisaldab üksikasjalikke andmeid haavatavuste kohta erinevate tarkvarasüsteemide ja rakenduste kohta. CVE Details pakub statistikat, graafikuid ja muud teavet, mis aitab turvaekspertidel ja organisatsioonidel mõista turvariskide ulatust ja olemust.

1.3.3 Common Vulnerability Scoring System (CVSS)

CVSS [10] on raamistik, mis võimaldab hinnata ja järjestada raporteeritud haavatavusi standardiseeritud ja korduvkasutataval moel. CVSS eesmärk on aidata võrrelda haavatavusi erinevates rakendustes, kasutades standardiseeritud, korduvkasutatavat süsteemi. Seda raamistikku kasutab CVE haavatavustele hinnangu andmiseks.

CVSS genereerib skoori vahemikus 0 kuni 10, põhinedes haavatavuse tõsidusele. Skoor 0 tähendab, et haavatavus on vähem märkimisväärne kui kõrgeim skoor 10. Kasutades CVSS-i haavatavuste prioriseerimiseks, on võimalik keskenduda kõige kriitilisematele.

CVSS-skoor [10] kombineerib mitmeid tegureid, et genereerida skoor. Need tegurid hõlmavad järgmist:

- **Ründevektor (*Attack Vector*)** - kuidas saadakse ühendust süsteemiga: võrk (*network*), kõrval (*adjacent*), kohalik (*local*), füüsiline (*physical*).
- **Ründe keerukus (*Attack Complexity*)** - kui lihtne on haavatavust ära kasutada: madal, kõrge.
- **Privileegide vajadus (*Privileges Required*)** - milliseid privileege on ründajal vaja enne rünnet: puudub (*none*), madal (tavaline kasutaja, *low*), kõrge (admin, *high*).
- **Kasutaja interaktsioon (*User Interaction*)** - kas kasutaja peab kuidagi osalema (*engage*) ründes: ei (*none*), vajalik (*required*).
- **Ulatus (*Scope*)** - valikud: muutumata (*unchanged*), muutunud (*changed*)
- **Konfidentsiaalsus (*Confidentiality*)** - ligipääs tundlikule infole: kõrge, madal, puudub

- **Terviklus (*Integrity*)** - andmete volitamata muutmise, andmete rikkumise või kustutamise võimalikkus: kõrge, madal, puudub.
- **Kättesaadavus (*Availability*)** - volitatud kasutajatele juurdepääsu keelamise võimalus: kõrge, madal, puudub.

Need tegurid kombineeritakse, et luua lõplik CVSS-skoor.

1.3.4 Common Platform Enumeration (CPE)

CPE [11] on standardiseeritud formaat, mida kasutatakse tarkvara ja selle versioonide identifitseerimiseks. NVD on loonud avaliku sõnastiku, milles CPE-de abil määratakse tarkvarale vastavad CPE väärtused, et kasutajatel oleks võimalus standardiseeritud kujul leida infot tarkvara kohta. Näiteks *"cpe:2.3:a:apple:safari:1.0:*:*:*:*:*:*:*"* tähistab Apple poolt arendatud Safari versiooni 1.0.

1.4 SwiftDependencyChecker

SwiftDependencyChecker [12] on käsurea programm, millega kontrollitakse Swifti paketihaldurite teekide turvavigasid. Seda saab ka integreerida programmeerimiskeskonnaga Xcode, mis võimldab haavatavusi kontrollida iga kord kui programmi ehitatakse (ingl *build*).

Programm on ülesehitatud järgmiselt [12]:

1. Tuvastatakse teegid. Seda tehakse failidel Podfile.lock (Cocoapods), Cartfile.resolved (Carthage) ja Package.resolved (Swift PM), kus on defineeritud teegid, koos versiooniga.
2. Leitud teekidele leitakse CPE vasted.
3. CPE vasted üritatakse sobitada NVD haavatavuste andmebaasi vastetega. Vaste leidmisel sisaldab CPE turvaviga.
4. Turvaveale üritatakse leida vastav teegi versiooni ja haavatavusi sisaldava teegi versiooni ühtivus. Selle leidmisel on tegemist haavatavust sisaldavad teegiga.
5. Otsitakse koodist import avalduse (ingl *statement*) järgi teegi kasutust.

Programmi jooksutades esimesel korral võtab see kauem aega, kuna alla tuleb laadida CPE sõnastik ja CocoaPods Spec hoidla (ingl *repository*) [12]. Teisel korral jooksutades on programm alla laadinud vajalikud failid ja analüüs võtab aega alla sekundi.

1.4.1 Teised tööriistad

Teised SwiftDependencyChecker-iga sarnase funktsionaalsusega tööriistad on näiteks Dependency-track[4], Bytesafe[5] ja OSSIndex[6]. Peamine funktsionaalsus, haavatavusi sisaldavate kolmandate osapoolte teekidest teada andmine, on neil sama. Erinevad nad üksteisest teotatud paketihaldurite ja lisafunktsioonide poolest.

1.5 Swift ja Java

Selles peatükis antakse ülevaade programmeerimiskeeltest, mida töös kasutatakse, ja nende erinevustest.

1.5.1 Swift

Swift [13] on kaasaegne ja programmeerimiskeel, mille on välja töötanud Apple eesmärgiga arendada rakendusi iOS, macOS, watchOS ja tvOS platvormidele. See esmakordselt avaldati 2014. aastal, asendades varasema Apple-i keele Objective-C, ja selle eesmärk oli pakkuda arendajatele paremat jõudlust, turvalisust ja lihtsamat kasutamist.

Swift on tüübiturvaline keel [13], mis kasutab kaasaegseid programmeerimisparadigmasid nagu objektorienteeritus, funktsionaalne programmeerimine ja protokollide orienteeritus. Sellel on puhas süntaks, mis muudab koodi lugemise ja kirjutamise lihtsaks ning võimaldab arendajatel luua keerukaid rakendusi.

1.5.2 Java

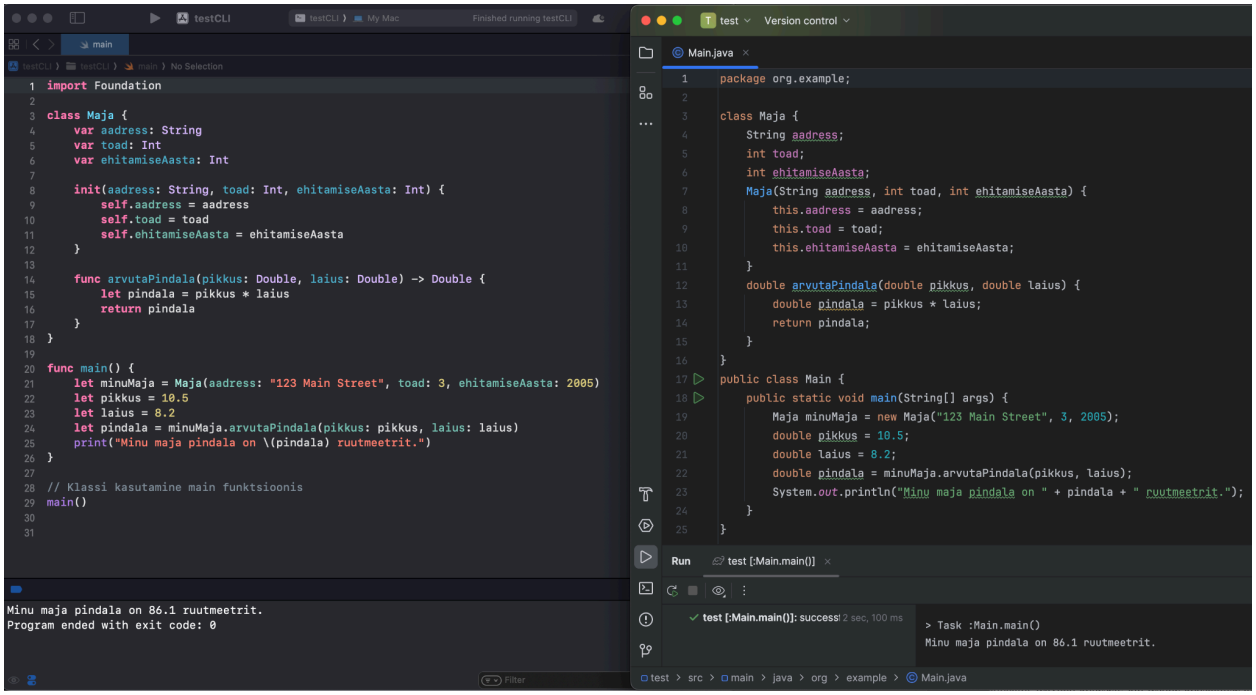
Java [14] on tüübiturvaline keel, mis kasutab objektorienteeritud programmeerimisparadigmat, kus programm on jaotatud väikesteks mooduliteks ehk klassideks. Üks Java peamisi eeliseid on selle võime töötada praktiliselt igal platvormil, kuna selle kood kompileeritakse esmalt

[4] https://github.com/DependencyTrack/dependency-track
[5] https://bytesafe.dev/
[6] https://ossindex.sonatype.org/

vahepealsele baidikoodile (ingl *bytecode*), mida saab käivitada Java virtuaalmasinas (JVM) erinevatel operatsioonisüsteemidel.

1.5.3 Erinevused

Üks suurimatest erinevustest nendel keeltel on platvormi tugi. Swift on seotud Apple-i ökosüsteemiga ja seda kasutatakse peamiselt iOS, macOS, watchOS ja tvOS rakenduste arendamisel. Kuigi Swift on avatud lähtekoodiga, siis veel ei ole kõik funktsionaalsused väljaspool macOS-i toetatud [15]. Joonisel 1 on välja toodud semiootilised erinevused, kus on kõrvuti sama kood nii Swift-is (vasakul) kui Javas (paremal) kirjutatud. Funktsioonide, muutujate ja klasside defineerimine ning funktsioonide väljakutsumine on väga sarnane. Näha on väikesed erinevused näiteks, kummal pool muutuja nime on tema tüüp defineeritud ja kuidas käib uue klassi initsieerimine.



Joonis 1. Swifti ja Java võrdlus

2. Meetod

2.1 Olemasolevate tööriistade uurimine

Olemasolevaid kolmandate osapoolte teekide haavatavusi kontrollivate tööriistu hakatakse uurima guugeldades populaarsemaid valikuid. Nendest tehakse tabel, uurides nende kohta, kuivõrd paljusid haldureid nad teotavad, milliseid haavatavusi kasutavad, kuna neid viimati uuendati, kas on tasulised. Tabeli põhjal otsustatakse, milliseid tööriistu hakatakse rohkem süvitsi uurima.

2.2 SwiftDependencyChecker kolimine teise keelde

Antud peatükis kirjeldatakse programmi SwiftDependencyChecker kolimist Swift programmeerimiskeelest Java programmeerimiskeelde.

2.2.1 Kolimise põhjused

Hetkel on programm kirjutatud programmeerimiskeeles Swift, mis on tihedalt seotud Xcode-ga. Kuigi Swift on saadaval nii Windowsi kui Linuxi peal, siis selle funktsionaalsus on piiratud. Eelkõige kasutatakse Swifti koos Xcode-ga Apple süsteemidele programmide loomiseks, aga võib tekkida olukord, kus ei ole võimalik ühtegi Apple seadet kasutada.

Selline olukord võib tekkida kui soovitakse programmi kasutada serveris kui kood on juba üles laetud. Teise selline situatsioon võib tekkida kui on soov kontrollida kellegi teise programmi. Sellisel juhul ei pruugi kontrollijal olla olemas mõnda macOS-iga seadet ja selle tõttu ei saa veenduda, et programm ei sisalda mõnda kolmandate osapoolte teekidest põhjustatud haavatavust.

2.2.2 Kolimise protsess

Kolimisel ühest keskkonnast ja keelest teise on eesmärk jätta funktsoonid, klassid, muutujad ja ülesehituse võimalikult sarnaseks. See lihtsustaks hiljem testimist, koodist arusaamist ja edasiarendust.

Kolimist alustatakse main.swift failist, milles antakse käsurea rakendusele ette parameetrid, millega tööd tegema hakatakse ja kutsutakse välja funktsioonid, millega hakatakse haavatavusi kontrollima. Peamine programmi funktsionaalsus on lüliti (ingl *switch*) sees. Sellest tulenevalt on võimalik juhu (ingl *case*) haaval hakata programmi kolima. Juhud ei ole täielikult iseseisvad, mis tõttu tuleb esimeste juhtudega defineerida klasse, funktsioone ja muutujaid, mida kasutatakse ka hiljem.

Swift-i kood antakse ette ChatGPT-le ja käsu "*Translate this code from Swift to Java:* [kood]" abil saadakse esialgne versioon kolitud koodist. Seejärel kontrollitakse tulemus, vajadusel kästakse ChatGPT-l teha muudatusi. Kui tehisintellekt sellega hakkama ei saa, tuleb teha muudatusi manuaalselt.

Swift-i koodis kasutatakse ka teeke, mida Java-s ei ole. Funktsionaalsuselt sarnased teegid tuleb Javas asendada. Infot erinevate võimaluste kohta selleks saadakse ChatGPT käest, andes käsu teha loetelu erinevatest sarnase funktsionaalsusega teekidest, mida oleks võimalik Java-s kasutada. Selle jaoks kasutatakse käsklust "*Give me alternative Java dependencies for* [teegi nimi] *Swift dependency*". Vastuseid analüüsitakse vaadates nende dokumentatsiooni ja kui laialdaselt need levinud on.

2.2.2.1 ChatGPT kasutamine

Töös kasutatakse ChatGPT versiooni 3.5. Andes ChatGPT-le Swift-i funktsiooni ja käskides sellest teha Java funktsioon, saab ta sellega hakkama mõningate puudustega. Tavaline lihtne süntaksi teisendus töötab tavaliselt hästi, aga kui on ise tehtud klassid, kolmandate osapoolte teegid, siis ei suuda tihti ChatGPT rahuldavat vastust anda. Sellisel juhul tuleb põhjalikumalt uurida Swift-is kirjutatud funktsionaalsus ja süntaksit, et seda teisendada Javasse.

2.2.3 Testid

Kui kolimine on tehtud on võimalik tulemust testida komponenditestidega (ingl *unit testidega*), luues sama sisendiga testid funktsioonidele ja oodates sama tulemust. Kuna funktsioonide funktsionaalsus ei muutu on võimalik paralleelselt tulemusi võrrelda. Kuna programmi SwiftDependencyChecker-it on juba testitud, siis kui kolitud programmi funktsionaalsus vastab sellele, siis programm töötab korrektselt.

2.3. SwiftDependencyChecker juurdearendus

Selles peatükis kirjeldatakse juurdearenduse meetodit.

2.3.1 Eesmärk

Leida turvaveaga teegi kasutamine asukoht, mitte ainult import avalduse (ingl *statement*) kaudu, ja anda nendest teada kasutajale.

2.3.2 SourceKit

SourceKit [16] on tööriist , mis pakub arendajatele võimalust manipuleerida Swifti lähtekoodi ja on osa Apple-i arenduskeskkonnast Xcode-ist. SourceKit pakub funktsioone nagu süntaksikontroll, automaatne lõpetamine, tarkvara analüüs ja muud funktsioonid, mis aitab kiiremini ja tõhusamalt töötada. See pakub ka arendajatele tuge erinevate programmeerimiskeelte, nagu Swift ja Objective-C, jaoks. SourceKit on oluline osa Xcode-ist, mis aitab kaasa sujuvale arendusprotsessile Apple-i platvormidele mõeldud rakenduste loomisel.

Avalik dokumentatsioon SourceKit-il puudub, kuna on mõeldud eelkõige Apple-s siseliselt kasutamiseks. Selle tõttu on keeruline selle tööriista funktsionaalsust implementeerida. Võimalik on näha logisid, kus Xcode kasutab SourceKit-i funktsionaalsust ja selle järgi implementeerida samu käske oma programmi.

2.3.2 Sõne järgi tuvastamine

Juhul kui SourceKit-i kasutades ei saada tööle on võimalus ka nime järgi leida potentsiaalseid haavatavusega seotud funktsioone ja muutujaid. Parsides import avaldusele järgnevat stringi on võimalik leida potensiaalseid kasutuskohti. Kasutaja saaks tõenäoliselt ka valenegatiivseid vasteid sellisele päringule, aga tegemist oleks lisafunktsionaalsusega, mille kasutaja saab soovi mitte sisse lülitada. See lahendus töötaks ka olenemata kasutatavast programmeerimiskeskonnast, mis võimaldaks seda funktsionaalsust kasutada sõltumata platvormist.

4. Tulemus ja arutelu

Tööga sooviti analüüsida erinevaid kolmandate osapoolte teekide kontrollimise tööriistu, kolida programm SwiftDependencyChecker teise keelde ja lisada funktsionaalsust.

4.1 Tööriistade analüüs

Selles peatükis antakse ülevaade olemasolevatest teekide kontrollimiste tööriistadest. Uurides, millised on nende põhilised funktsioonid, milliseid raamistikke need toetavad ja milliseid andmebaase kasutavad. Esmalt valmis tabel lingil https://docs.google.com/spreadsheets/d/1gcAzHrCWvZJhahBuap15r5Db7YBkyfnt2PujhUshFAA/edit?usp=sharing. Sealt valiti tähtsamad tööriistad, mida analüüsitakse põhjalikumalt järgmistes peatükkides.

4.1.1 Dependency-track

Dependency-track [17] arendajaks on OWASP (Open Worldwide Application Security Project), mis on mittetulunduslik fond, mille ülesandeks on parandada tarkvara turvalisust. Programm on avatud lähtekoodiga ja toetab Cargo, Composer, Gems, Hex, Maven, npm, CPAN, NuGet ja Pypi teekide haldureid. Turvaaukude kontrollimiseks kasutavad nad NVD, GitHub Advisories, Sonatype OSS Index, Snyk, OSV ja VulnDB andmebaase.

4.1.2 Bytesafe

Bytesafe [18] on tasuliste võimalustega teekide kontrollimise rakendus. See toetab Maven, NPM, NuGet, Pnpm, Pip, Yarn ja Gradle raamistikke. Andmebaasid, kust kontrollitakse turvavigu on NVD ja GitHub Security Advisories. Lisaks pakub programm veel ka litsentside sobivuse kontrolli, millega tehakse kindlaks, kas antud kolmandate osapoolte teeki võib näiteks ärilistel eesmärkidel kasutada.

4.1.3 OSSIndex

OssIndex [19] pakub võimalust kontrollida turvavigu Cargo, CocoaPods, Composer, Conan, Conda, CRAN, Go, Maven, NPM, NuGet, PyPI, RPM, Ru

byGems, Swift komponentidest. Nende andmebaasideks on NVD, GitHub Advisories ja leitud haavatavustest on võimalik neile e-kirja teel teada anda.

4.1.4 Dependabot

Dependabot [20] on GitHub-i sisse ehitatud tööriist, mis kontrollib automaatselt projektis olevate sõltuvuste haavatavusi ja töötab enamuste pakettide halduritega. Andmebaasina kasutatakse GitHub Security Advisories andmebaasi. Dependabot-i on võimalik seadistada, et see teeks ise ka tõmbetaotlusi (ingl *pull request*), kui programm leiab mõne turvavea. Samuti on võimeline Dependabot tõmbetaotlusi ka ise harusse liitma (ingl *merge*).

4.1.5 Snyk

Snyk [21] pakub võimalust lisaks kolmandate osapoolte teekide haavatavuste kontrollimisele ka koodianalüüsi. Sarnaselt Dependabot-ile on võimalik teha tõmbetaotlusi, mis uuendab pakette automaatselt. Snyk on ühilduv enamuste pakettihalduritega ja omab enda andmebaasi haavatavuste kohta.

4.1.6 Analüüsi kokkuvõte

Analüüsides erinevaid populaarsemaid teekide turvalisust kontrollivaid tööriistu saadi teada, et ettevõtted ja organisatsioonid on loonud palju lisafeatuure (vaata Tabel 1), et eristuda teistest. SwiftDependencyChecker eristub teistest oma lihtsuse poolest. Avalikud andmebaasid, kuhu saab teha päringuid, võimaldab lihtsa vaevaga saada teada, kas kolmanda osapoole teegil on haavatavusi. Protsessi lisab keerukust kui teekide nimed ja versioonid tuleb välja lugeda failidest, mis on erinevatel paketihalduritel erinevalt defineeritud. Kuid ka see ei ole tehniliselt väga keerukas, kui teada, millise ülesehitused on konkreetsed failid.

Tabel 1. Analüüsitud tööriistade tabel

Tööriista nimi	Toetatud paketihaldurid	Eristuvad featuurid
Dependency-track	Cargo, Composer, Gems,	Automaatsed teavitused,

	Hex, Maven, NPM, NuGet, Pypi, CPAN	litsentside kontroll
Bytesafe	Npm, NuGet, yarn, pnpm, pip	Litsentside kontroll
Dependabot	Bundler, Cargo, Composer, Docker, Hex, Go, Gradle, Maven, npm, NuGet, pip, yarn	Automaatsed tõmbetaolused, harusse liitmised, automaatsed teavitused
Snyk	Cargo, cocoapods, Composer, Go, hex, Maven, npm, NuGet, pip, pub, RubyGems	Automaatsed tõmbetaolused, automaatsed teavitused

Ettevõtted pidanud looma lisafeatuure, mis eristaks neid konkurentidest. Ainukesena täielikult tasuta on Dependency-track. Ülejäänute funktsionaalsus on piiratud kasutades tasuta versiooni ja omavad tellimuspõhist ärimudelit. See tähendab, et nende programmide arendamiseks on meeskonnad inimestega, kes neid arendavad, mis võimaldavad neil luua keerulise lisafunktsionaalsusi, näiteks integreerides oma teenuseid teistesse platvormidesse.

Lihtsam funktsionaalsus, mis puudus SwiftDependencyChecker-il oli turvaveaga teekide funktsioonide väljakutsumise tuvastamine. Samuti ei olnud seda funktsionaalsust tööriistadel Snyk ja Dependency-track.

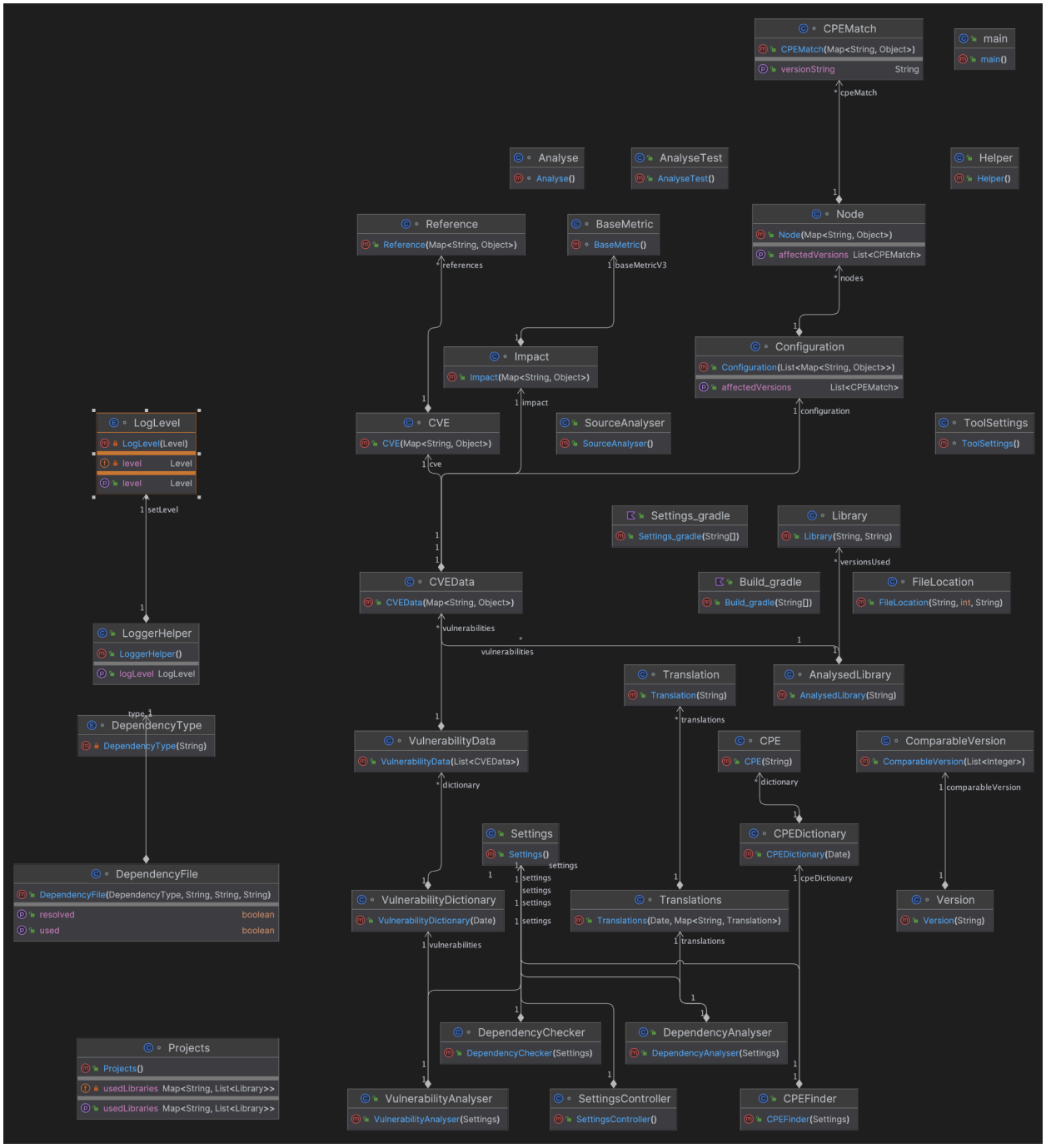
4.1.7 Analüüsi järeldus

Automaatne tõmbetaotluste tegemine on üks suuremaid lisafunktsioone, mis osadel haavatavusi kontrollivatel programmidel on. Selle implementeerimine, aga nõukas tõenäoliselt väga suuremahulist arendustööd. Lihtsam funktsionaalsus, mis SwiftDependencyChecker-il veel ei ole on haavatavusi sisaldavate teekide reaalne kasutamine. Hetkel kontrollitakse SwiftDependencyChecker-i puhul ainult import avalduse järgi kasutamist. Sellega saadakse ka

vale positiivseid vasteid, kuna pole kindel, kas haavatavust sisaldava teegi haavatavat funktsiooni kasutatakse. Seda automatiseerida oleks keeruline, aga kui anda kasutajale teada kohast, kus kasutatakse haavatava teegi funktsiooni, siis kasutaja saaks ise uurida, kas teeki on vaja uuendada.

4.2 Kolimine

SwiftDependecyChecker koliti keelest Swift üle keelele Java. Joonisel 3 on näha, et kokku koliti 34 klassi, lisaks nende meetoditega. Kolimisele aitas ChatGPT. Väga hästi sai sellega lihtsaid süntaksi teisendusi ühest programmeerimiskeelest teise teha. Ilma sellest oleks kolimise osa võtnud kordades rohkem aega. Probleeme tekkis kolmandate osapoolte teekide kasutamisega. Selle juures ei suutnud ChatGPT alati õige tulemuseni jõuda. Väga kindlalt andis ta valesid vastuseid ja üritades teda parandada andis ikka samu valesid vastuseid. Sellistel juhtudel tuli käsitsi lahendus leida, lugedes nii Swift-i poolset kui Java-s kasutatava kolmanda osapoole teegi dokumentatsiooni.

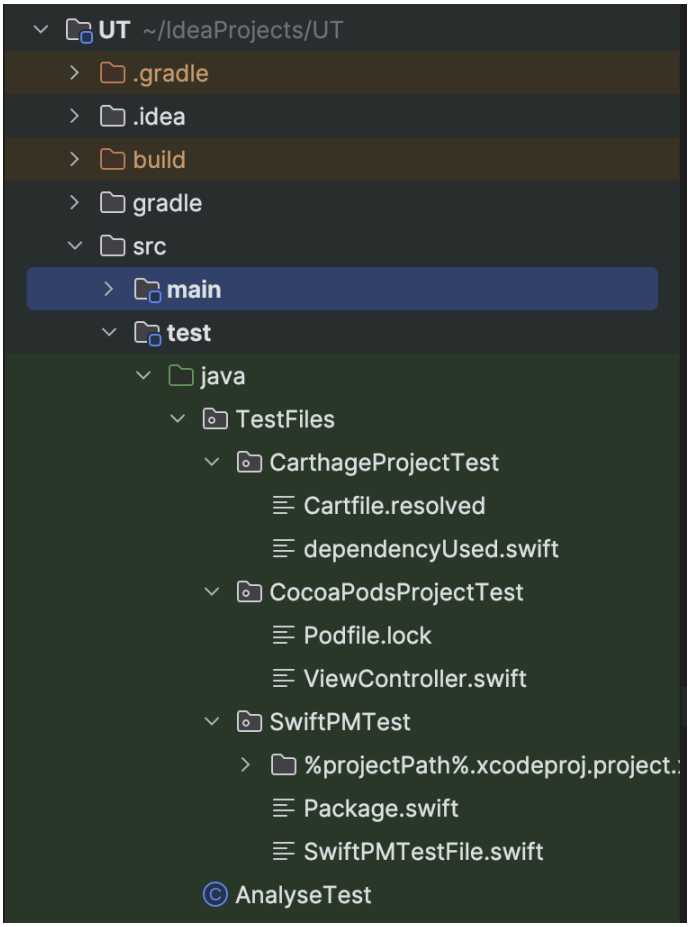


Joonis 3. Projekti diagram

Lisaks ChatGPT valede vastuste andmisele tekkis probleeme ka API-de kasutamisega. Vahepeal oli muutunud NVD API, mille tõttu ei töötanud ka originaalne Swifti kood. Muudetud oli nii API välja kutsumise parameetreid kui ka vastuse skeemi (ingl *schema*). Klasside süsteem on SwiftDependencyChecker-il üles ehitatud nii, et API vastab päringu peale .json faili, millest

loetakse sõnastikke väärtused klassidesse. Selle tulemusena tuli muuta klasside väljade väärtustamise loogikat. Tulevikus peaks vahetama välja ka vanade muutujate nimed, mis hetkel suurel hulga oma nime saanud API vastest sõltuvalt.

Tööriista testimiseks on loodud failid kausta nimega *test*. Joonisel 4 on välja toodud failistruktuur, kus on SwiftPM, CocoaPods, Carthage näidisprojektid. Neil on õige süntaksiga paketihaldurid, milles on defineeritud haavatavusi sisalduvad teegid. Samuti on üks .swift näidisfail, milles on välja kutsutud haavatavust sisaldavat teeki. Jooksutades programmi ja seades *-path* väärtuseks testprojektide asukoha on võimalik kontrollida programmi töötamist.



Joonis 4. Testide kaustad

4.2.1 Kasutatud Java kolmandate osapoolte teegid

Swiftis on kasutatud kolmandate osapoolte teekidena GzipSwift[7]-i, mida kasutatakse failide lahtipakkimiseks (ingl *unzip*) ja Swift Argument Parser[8]. GzipSwift-i kasutatakse allalaetud .gz laiendiga faili lahtipakkimiseks ja Swift Argument Parser-it kasutatakse CLI programmi tegemiseks. Gzip on Java.util.zip. sisseehitatud, CLI programmi tegemiseks kasutatakse Javas Picocli teeki. Siin sobib Picocli hästi kuna on süntaksi poolest väga sarnane Swift Argument Parseriga. Lisaks on Javas kasutatud teeke:

- **Gson** - Kasutatud json failide lugemiseks ja kirjutamiseks. Valitud oma lihtsa süntaksi ja laialdase kasutuse pärast.
- **Javatuples** - Javas ei ole sisse ehitatud klassi, mis paarist suuremat "listi" teeks.

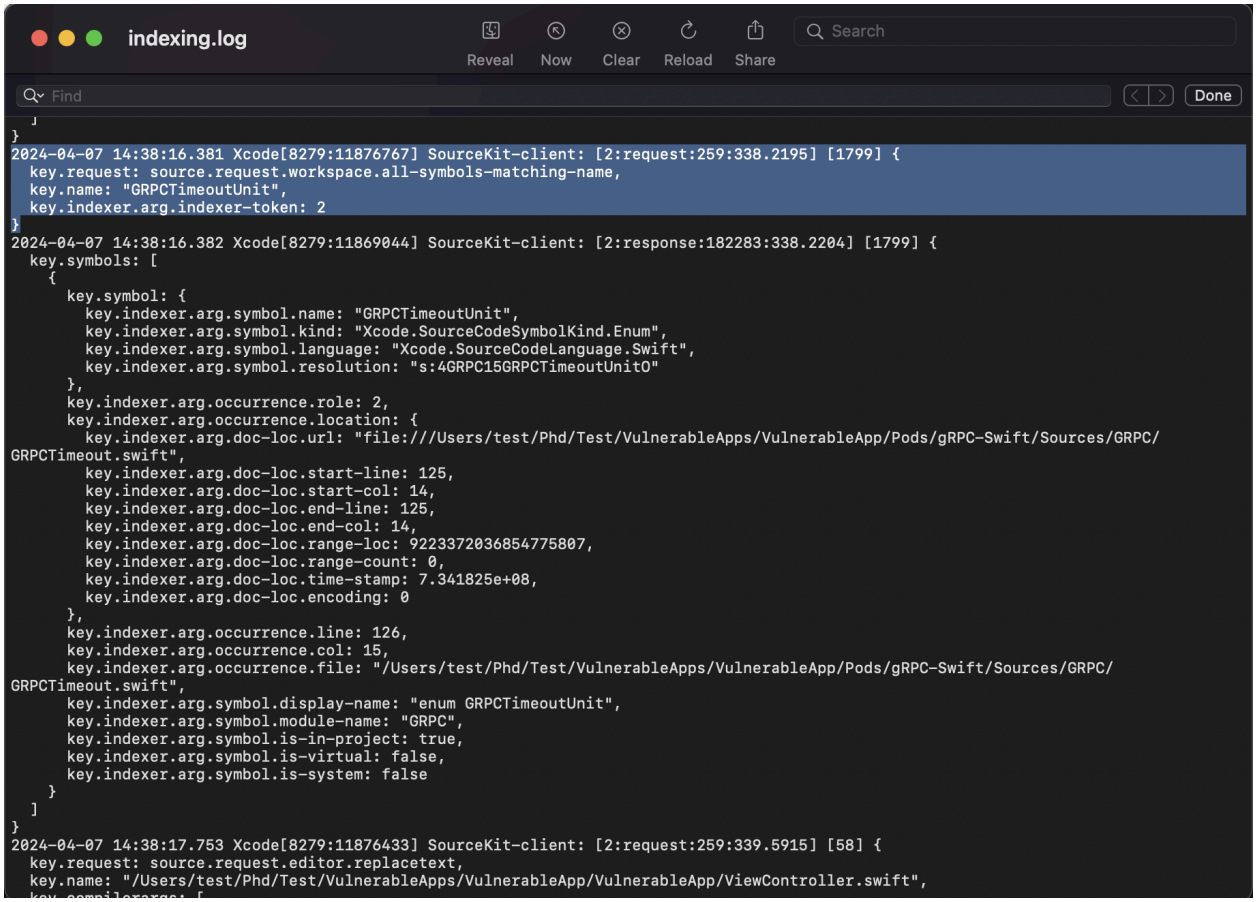
4.3 Juurdearendus

Juurdearenduse puhul realiseerus sõne parsimise meetod. SourceKit-i kasutamisel leiti küll käsk, millega oleks võimalik teada saada, millisest teegist on päris funktsioon, mida kasutatakse, aga SourceKit ei lase seda käsku eraldiseisvalt kasutada.

4.3.1 SourceKit-i kasutamise analüüs

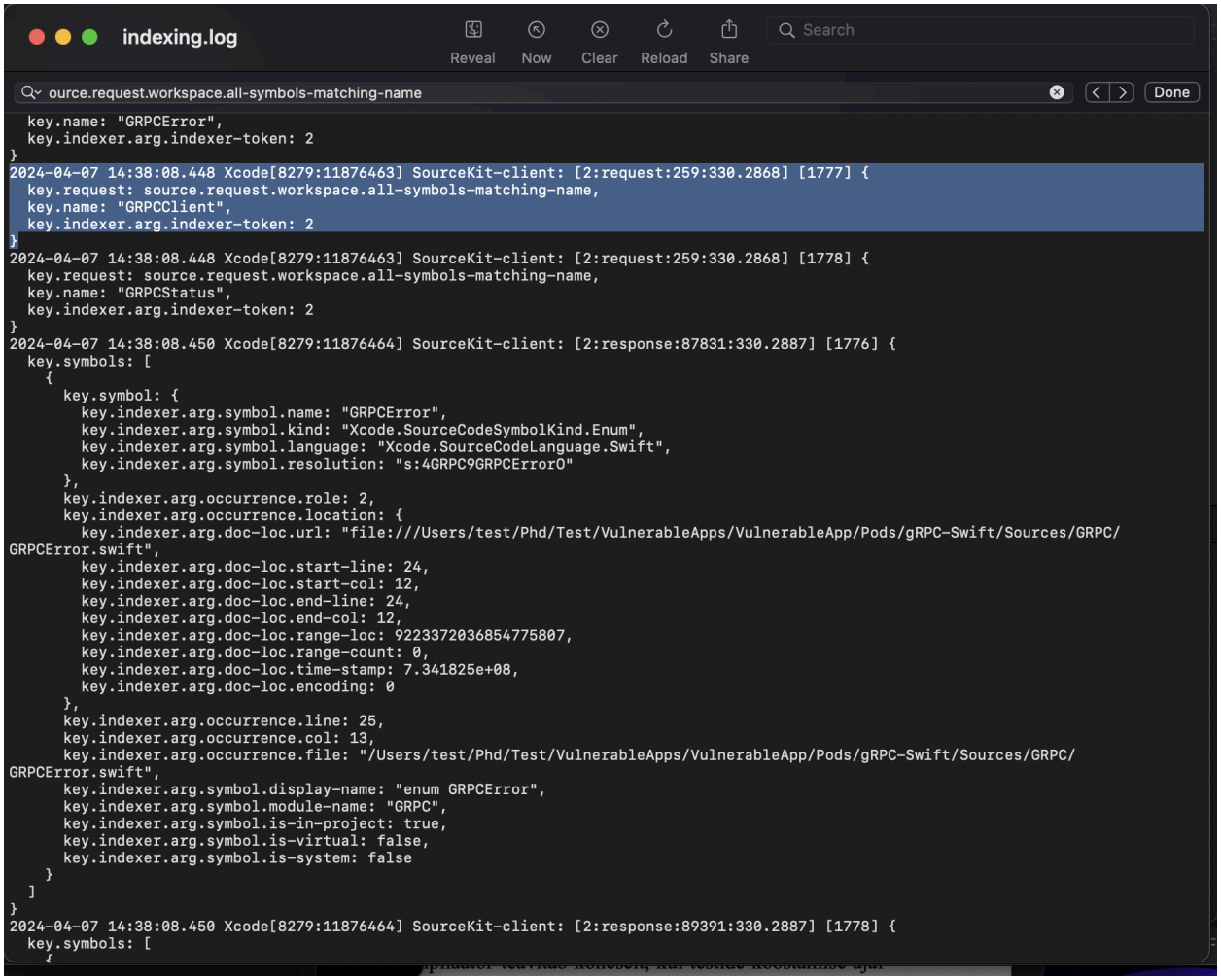
Xcode kasutab taustal SouceKit-i, et indekseerida koodi, tehes selleks SourceKit-ile päringuid. Vaadates SourceKit-i logisid, jooniselt 5, on näha, et Xcode kasutab käsku (aktiivseks tehtud osa), millega on võimalik näha kasutatud meetodi teeki. Päringuga *key.request: source.request.workspace.all-symbols-matching-name* ja nimega *key.name: "GRPCTimeoutUnit"* väljastab SourceKit rea *key.indexer.arg.symbol.module-name: "GRPC"*. See tähendab, et element *GRPCTimeoutUnit* on osa teegist *GRPC*. Kontrollides iga sellise elemendi *key.indexer.arg.symbol.module-name* väärtust ja sobitades seda meile teada olevate haavatavasi sisaldavate teekidega oleks meil võimalik välja tuua kõik teegi kasutuskohad.

[7] https://github.com/1024jp/GzipSwift
[8] https://github.com/apple/swift-argument-parser



Joonis 5. Xcode päring SourceKit-ile

Eelnev logi joonisel 5 oli saadud SourceKit-i ja Xcode omavahelisest päringute tegemisest. Võimalik on ka ise SourceKit-ile käske anda. Joonisel 6 on näide sellest. Saadetud on päring (aktiivseks tehtud), aga käsitsi päringut tehes SourceKit vastust ei anna. Arvatavasti on vaja mõnda päringut, mis eelneks tehtud päringule. Kuna dokumentatsioon selle tööriistal puudub, siis ei ole teada, millised on eeldused, et seda päringut saaks teha.



Joonis 6. Ebaõnnestunud päringud SourceKit-i

4.3.2 Sõne parsimise tulemus

Sõne parsimise meetod hakkab andma vale negatiivseid ja vale positiivseid vastuseid, selle pärast on see realiseeritud ka valikulise parameetrina. Jooniselt 7 on näha, et selle vaikeväärtus sellele valikule on väär. Selleks, et see meetod ka välja kutsutakse peab lisaks *findVulnerableDepenencyNameUsage* tõesele väärtusele olema valitud *Action all*, mis on ka vaikeväärtuseks.

```
@CommandLine.Option(arity = "1", names = {"-f", "--findVulnerableComponentUsage"}, description = "Match vulnerable " +
        "dependency name in file.")
boolean findVulnerableDependecyNameUsage = false;
```

Joonis 7. Loodud lisaparameeter

Jooniselt 8 on näha kogu edasiarenduse loogika. Funktsioonis *searchInSwiftFiles* rakendatakse rekursiivset *search* funktsiooni. Kui *search* tekitab nimekirja etteantud kaustas olevates failides ja leides Swift faili uuritakse selle sisu. Kui etteantud fail on kaust, siis vaadatakse selle kausta sisu rekursiivselt. Swift-i failides kontrollitakse, kas fail omab rida *import* koos etteantud turvaviga sisalduva teegi nime. Kui jah, siis otsitakse kõik etteantud nime sisaldavad kohad failist üles ja logitakse need kasutajale. Joonisel 9 on näha logimist. Logitakse turvaveaga teek, fail, milles turvaviga kasutatakse, ja kasutamise asukoht failis.

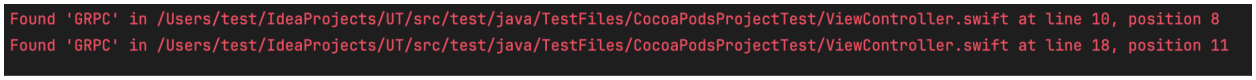
```java
public static void searchInSwiftFiles(String path, String searchString) {
    File directory = new File(path);
    if (!directory.isDirectory()) {
        LoggerHelper.log(LogLevel.ERROR, "Invalid directory path.");
        return;
    }
    search(directory, searchString);
}

private static void search(File directory, String searchString) {
    File[] files = directory.listFiles();
    if (files != null) {
        for (File file : files) {
            if (file.isDirectory()) {
                search(file, searchString);
            } else if (file.getName().toLowerCase().endsWith(".swift")) {
                try {
                    if (containsImport(file, searchString)) {
                        searchInFile(file, searchString);
                    }
                } catch (IOException e) {
                    LoggerHelper.log(LogLevel.ERROR,"Error reading file: " + file.getName());
                }
            }
        }
    }
}

public static boolean containsImport(File file, String searchString) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        String line;
        while ((line = reader.readLine()) != null) {
            if (line.contains("import") && line.contains(searchString)) {
                return true;
            }
        }
    }
    return false;
}

public static void searchInFile(File file, String searchString) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        String line;
        int lineNumber = 0;
        while ((line = reader.readLine()) != null) {
            lineNumber++;
            if (line.contains(searchString)) {
                int position = line.indexOf(searchString);
                LoggerHelper.log (LogLevel.INFO, "Found '" + searchString + "' in " + file.getAbsolutePath()
            }
        }
    }
}
```

Joonis 8. Edasiarenduse implementeerimine



Joonis 9. Edasiarenduse väljund

4.4 Edasiarenduse võimalused

Järgnevalt on välja toodud mõned mõtted, mida võiks tulevikus selle projekti raames edasi arendada:

1) Praegu jäi SourceKit-i lahendus implementeerimata. See võimaldaks oluliselt täpsemalt anda kasutajale tagasiside tema haavatavustega kasutatavate teekide kohta. Isegi kui see oleks praegu ainult macOS põhine, siis see funktsionaalsus olekski eelkõige kasulik koodi kirjutades.

2) Selles töös ei analüüsitud põhjalikumalt võimalust teha tõmbetaotlusi teekide uuendamiseks ja nende automaatset harusse liitmist. Võimalik, et tegemist ei ole nii keerulise funktsionaalsusega kui algselt arvati ja oleks võimalik vähese vaevaga see implementeerida.

Samuti tuleb projekti ajakohasena hoida. Nagu ka töö käigus välja tuli, siis muutuvad API-d ja võivad muutuda ka paketihaldurid. Neid tuleks pidevalt jälgida, et programm oleks kasutatav ka tulevikus. Hetkel on kõigi Swift-i paketihaldurid tugi SwiftDependencyChecker-il olemas, aga on võimalik, et neid tekib ajaga juurde või nad muudavad oma skeemi. See tähendab, et kui programmi pidevalt ei uuendata, võib selle funktsionaalsus väheneda. Et seda ei juhtuks tuleks mingi aja tagant jälgida, kas kõik funktsionaalsused töötavad.

Kokkuvõte

Bakalaureusetöö eesmärk oli analüüsida erinevaid kolmandate osapoolte haavatavusi kontrollivaid tööriistu, kolida rakendus SwiftDependencyChecker programmeerimiskeelelt Swift keelde Java ja lisada rakendusele funktsionaalsust. Olemasolevate tööriistade analüüsis selgitati lugejale erinevaid teemaga seotud mõisteid, anti ülevaade populaarsematest andmebaasidest, haavatavuste hindamisest ja standardiseeritud käsitlusest.

Kolimisega seotud peatükkides toodi välja kasutusjuhud, kus on laialdasema kasutusega programmeerimiskeelest Java kasu. Lisaks toodi välja kolimiseks kasutatud vahendeid ja teegid, mille abil säilitas programm oma funktsionaalsuse.

Funktsionaalsus, mis lisati oli haavatavusega teegi kasutuskoha leidmine. Toodi välja kaks erinevat viisi, kuidas seda implementeerida saaks ja üks nendes realiseeriti. Realiseerimata jäi lahendus, mis oleks SourceKit-i kasutanud, et leida haavatavusega teegi kasutamist. Selle asemel implementeeriti lihtsam lahendus, millega kontrollitakse haavatavusega teegi nime esinemist failis. Samuti selgitati, miks valiti just realiseerunud viis, ning mis eelised ja puudused sellel lahendusel on.

Viidatud kirjandus

[1] A. Papadopoulo, “Should Developers Use Third Party Libraries?,” *Scalable Path*, Mar. 28, 2020. https://www.scalablepath.com/back-end/third-party-libraries (Kasutatud: mai 15, 2024).

[2] “Third Party Libraries Definition,” *Law Insider*. https://www.lawinsider.com/dictionary/third-party-libraries (Kasutatud jan. 10, 2024).

[3] “Package management basics - Learn web development,” *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management (Kasutatud jan. 10, 2024).

[4] S. Brathwaite, “The Security Liabilities of 3rd Party Libraries.” https://www.softwaresecured.com/post/the-security-liabilities-of-3rd-party-libraries (Kasutatud jan. 12, 2024).

[5] “National Vulnerability Database (NVD),” *NIST*. https://www.nist.gov/programs-projects/national-vulnerability-database-nvd (Kasutatud jan. 10, 2024).

[6] github, “GitHub - github/advisory-database: Security vulnerability database inclusive of CVEs and GitHub originated security advisories from the world of open source software.,” *GitHub*. https://github.com/github/advisory-database (Kasutatud jan. 10, 2024).

[7] “Snyk Vulnerability Database,” *Snyk User Docs*. https://docs.snyk.io/scan-with-snyk/snyk-open-source/manage-vulnerabilities/snyk-vulnerability-database (Kasutatud mai 15, 2024).

[8] “What is a CVE?” https://www.redhat.com/en/topics/security/what-is-cve (Kasutatud jan. 10, 2024).

[9] “What is CVE and CVSS,” *Imperva Inc*. https://www.imperva.com/learn/application-security/cve-cvss-vulnerability/ (Kasutatud jan. 12, 2024).

[10] S. C. Leadership, “What is CVSS,” *Common Vulnerability Scoring System*. https://www.sans.org/blog/what-is-cvss/ (Kasutatud jan. 10, 2024).

[11] “NVD,” *CPE*. https://nvd.nist.gov/products/cpe (Kasutatud mai 15, 2024).

[12] K. Rahkema and D. Pfahl, “SwiftDependencyChecker: Detecting vulnerable

dependencies declared through cocoapods, carthage and swift pm.," *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems.* , pp. 107–111, mai 2021.

[13] "Swift," *Apple Developer*. https://developer.apple.com/swift/ (Kasutatud mai 15, 2024).

[14] "What is Java? - Java Programming Language Explained - AWS," *Amazon Web Services, Inc.* https://aws.amazon.com/what-is/java/ (Kasutatud mai 15, 2024).

[15] apple, "GitHub - apple/swift-foundation: The Foundation project," *GitHub*. https://github.com/apple/swift-foundation (Kasutatud mai 15, 2024).

[16] "Uncovering SourceKit - JP Simard," *Swift Developer*. https://www.jpsim.com/uncovering-sourcekit/ (Kasutatud mai 15, 2024).

[17] "About the OWASP Foundation," *OWASP Foundation*. https://owasp.org/about/ (Kasutatud dets. 05, 2023).

[18] "Bytesafe Documentation," *Bytesafe Documentation*. https://docs.bytesafe.dev/ (Kasutatud dets. 05, 2023).

[19] Inc. Sonatype, "Integrations," *Sonatype OSS Index*. https://ossindex.sonatype.org/integrations (Kasutatud dets. 06, 2023).

[20] "About Dependabot alerts," *GitHub Docs*. https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts (Kasutatud mai 15, 2024).

[21] "Snyk Vulnerability Database," *Find detailed information and remediation guidance for vulnerabilities and misconfigurations.* https://security.snyk.io/ (Kasutatud dets. 06, 2023).

Lisad

I. Loodud materjalid

Lõputöö raames Java keelde kolitud ja täiendatud programm on kättesaadav lehel:

https://github.com/ristovoor/dependency-checker/releases/tag/BsC

Sõltuvuste haavatavusi kontrollivate tööriistade tabel:

https://docs.google.com/spreadsheets/d/1gcAzHrCWvZJhahBuap15r5Db7YBkyfnt2PujhUshFAA/edit?usp=sharing

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Risto Voor,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Kolmandate osapoolte teekide kontrollimise tööriistade analüüs ja täiustamine,** mille juhendaja on Kristiina Rahkema, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Risto Voor

15.05.2024