JENSEN yrkeshögskola
TRÄNING FÖR VERKLIGHETEN

Software Composition Analysis

– Skanning av programvarupaket

Examensarbete

SYSÄ22M

VT 2024

Ola Persson Orator

Handledare: Ulf Eriksson

Sammanfattning

Detta examensarbete undersöker hur Software Composition Analysis (SCA) fungerar och vad den har för del i säkerhetsprocessen för att minimera möjliga sårbarheter i programkod. Ett par möjliga implementationer tas upp. Men jag drar slutsatsen att det inte är vilket SCA-verktyg som är det viktigaste, utan att SCA tillsammans med en i övrigt heltäckande säkerhetsprocess är den bästa lösningen.

Innehåll

Inledning

I dagens samhälle är det viktigt att programvara är säker, då vi använder mjukvara till i princip allt vi gör i vårt dagliga liv. Sårbarheter i programvara kan utnyttjas och skapa allvarliga störningar, därför är det viktigt att alla delar av utvecklingsprocessen tänker på säkerhet. En av åtgärderna är Software Composition Analysis (SCA) som kontrollerar att det inte finns kända säkerhetshål i programvarupaketen vi använder.

Detta arbete syftar till att utforska användningen av SCA för att minimera säkerhetshål i mjukvara. Genom att analysera olika metoder, verktyg och praxis inom SCA, kommer jag att undersöka hur man bäst gör för att integrera SCA i den befintliga utvecklingsprocessen.

Under min LIA implementerade jag bland annat en process som utför SCA och skickar uppgifterna till ett verktyg för hantering av sårbarheter i mjukvara.

Syfte

Syftet med detta arbete är att undersöka hur man kan arbeta med Software Composite Analysis (SCA) för att minimera säkerhetshål i använda programvarupaket, även kallade bibliotek. Målet är att förstå hur SCA kan integreras i en CI/CD-pipeline och vilken information utvecklarna behöver för att identifiera och åtgärda potentiella problem. Dessutom vill jag undersöka vad som krävs för att optimera denna process.

Frågeställningar

1. Hur kan man upptäcka sårbarheter i programvarupaket som påverkar ens egen kod?
2. Vad kan man göra för att skydda sig mot åldrande programvarupaket?
3. Vilka verktyg finns tillgängliga för att övervaka nya sårbarheter och identifiera dem i den egna programvaran?
4. Hur kan Software Composition Analysis (SCA) integreras i utvecklingsprocessen så att det blir en naturlig del av den?

Metod

Litteratur

Jag kommer framför allt att använda mig av internet för att söka information, eftersom säkerhet är en färskvara. För att använda pålitliga källor kommer jag fokusera på organisationer som är erkända och välkända av majoriteten i säkerhetssammanhang. I huvudsak kommer jag att titta på organisationer med öppen källkod eftersom det är lättare att verifiera att informationen som anges är korrekt. Nackdelen med att hämta information från Internet är att all information måste verifieras.

Därför tar vi en närmare titt på RADAR [Layola Marymount University] som ett verktyg för att utvärdera internetkällor. Jag hänvisar till en andrahandskälla av "RADAR: An approach for helping students evaluate Internet sources." [Mandalios]. RADAR är en förkortning för:

- Releveance (Relevans): Att bedöma hur relevant källan är för ämnet eller frågan som skall undersökas.
- Authority (Auktoritet): Utvärdera författaren och/eller utgivarens auktoritet och expertis inom ämnet som undersöks.
- Date (Datum): När publicerades källan eller uppdaterades den senast? Äldre information kan vara föråldrad eller inte längre vara relevant för undersökningen.
- Accuracy (Noggrannhet): Bedöma om informationen är korrekt och pålitlig. Finns det bevis på påstående och stöds informationen av andra källor.
- Rationale (Motivering): Undersök syftet och/eller motivet bakom informationen.

Genom att använda RADAR så kan jag få en mer medveten och kritisk inställning till de källor jag använder på internet och ökar därmed kvaliteten på undersökningarna och detta arbete.

Observation

Jag kommer att använda mig av information som jag observerade under min LIA-period. Företaget som jag praktiserade för ville införa SCA i sin pipeline och gav mig en uppgift att lösa. Uppgiften var klart definierad och jag höll mig till definitionen tills den visade sig vara mindre lämplig. Att använda sig av

observation är en fördel för jag har förstahandsinformation och det är färsk information. En nackdel är att den kan vara vinklad, vilket jag kommer att ta hänsyn till. Framförallt kommer jag att ha med mig implementationen av OWASP Dependency-Check och integreringen till OWASP DefectDojo. Jag kommer även att referera till återkopplingar på hur CI/CD fungerar för utvecklarna och hur införandet påverkade dem.

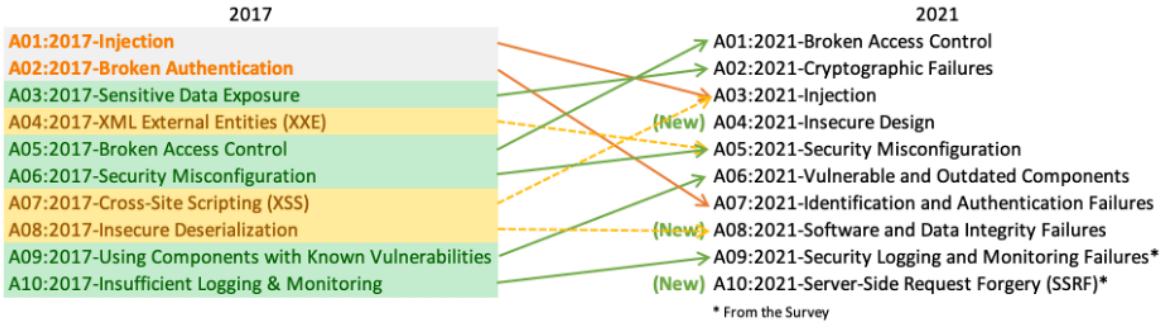
Resultat

Litteratur

I den här sektionen väljer jag att arbeta med metoden RADAR för att säkerställa att jag har kritisk inställning till den information som finns på internet och därmed få bra kvalitet på mina källor. Det är inte en uttömmande utredning av varje källa utan jag fokuserar på att ge övergripande information om de olika källorna så att jag till slut kan besvara frågeställningarna.

OWASP en introduktion

Open Web Application Security Project [OWASP] är en gemenskap av säkerhetsexperter som arbetar mot säkrare programvara. OWASP har en lista på de 10 mest kritiska sårbarheterna i webbapplikationer som de har identifierat.



Figur 1 [OWASP] Owasp top 10, 2021

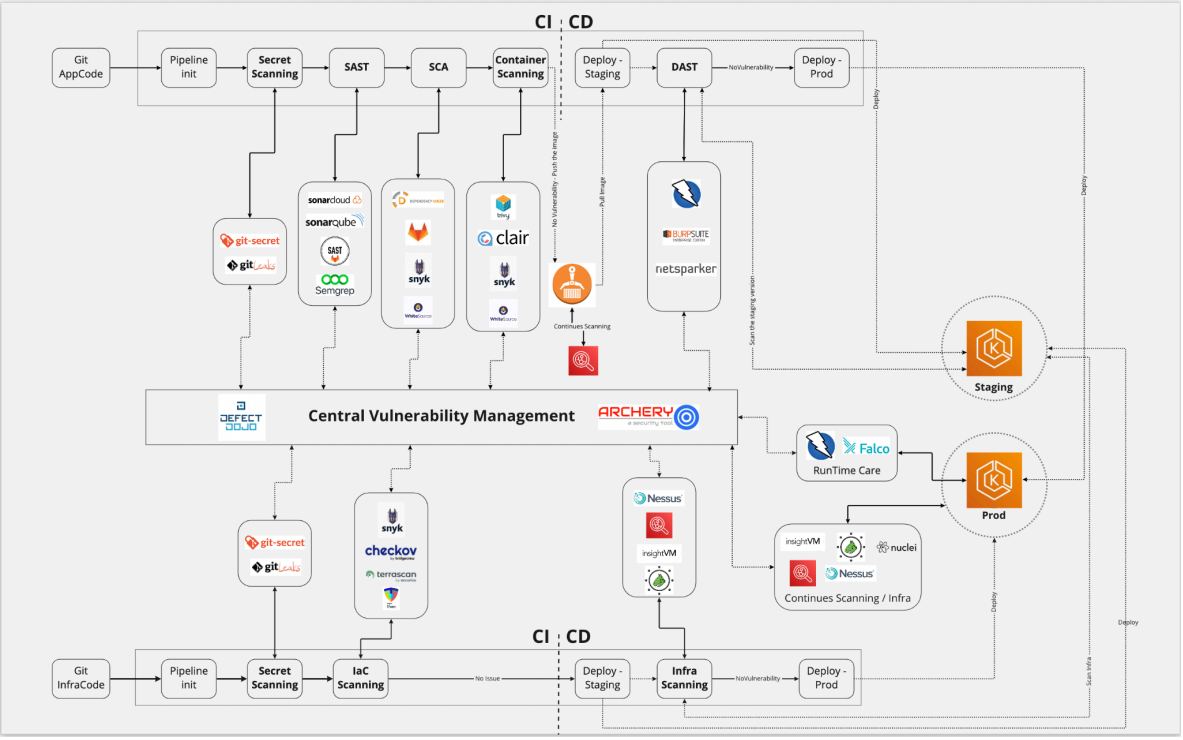
I 2021 års topplista hamnar "A06:2021 – Sårbara och Föråldrade Komponenter" på plats 6. I denna kategori så tar man upp programvarupaket och dess sårbarheter. Common Vulnerabilities and Exposures [CVE] är en allmänt accepterad standard [MITRE CVE] för att identifiera och namnge sårbarheter i mjukvara. Varje CVE-identifierare är unik för den specifika sårbarheten, vilket gör att alla har en gemensam referenspunkt när man diskuterar en specifik sårbarhet. Det minimerar risken att man diskuterar olika säkerhetsproblem. CVE-listan skapades i huvudsak av MITRE, år 1999 [MITRE CVE], för att främja cyberförsvar. Förutom MITRE kan medlemmar av CVE Numbering Authorities [CNAs] lägga till CVEer på CVE-listan. CNAs är en internationell grupp av företag och forskare från ett flertal länder som administreras av MITRE.

Förutom att OWASP sammanställer sårbarhetslistan Top Ten, så har de även projekt för utbildning om säkerhet, verktyg för säkerhet och sammanställningar för hur man bäst arbetar med säkerhet. Detta för att hjälpa utvecklare och säkerhetsexperter att identifiera och hantera säkerhetsrisker i webbaserade

applikationer och för att öka medvetenheten och främja bästa praxis inom utveckling och säkerhet.

Software Composition Analysis

SCA är en metod för att utvärdera och hantera mjukvarupaket och deras beroenden i ett projekt. Nedan följer en bild om hela hanteringen av säkerhetsprocessen som kan finns i CI/CD för ett mjukvaruprojekt och man kan se att SCA är en del i den.



Figur 2 [OWASP] Guidelines for DevSecOps

Målet med SCA är att säkerställa att användningen av komponenter i ett projekt är säkra och följer bästa praxis, det inkluderar att upptäcka och åtgärda eventuella säkerhetsbrister eller licenskompatibilitetsproblem [SCA]. Det rekommenderade sättet att lösa OWASP "A06:2021 – Sårbara och Föråldrade Komponenter".

Några av de vanligaste riskfaktorerna för projekt som SCA försöker lösa är enligt Steve Springett på OWASP Foundation [Springett]:

- Inventering av komponenter gör att man har koll på alla komponenter ett projekt använder och därmed kan man göra riskbedömning. Utan denna faktan är det nästan omöjligt att utföra SCA.
- Komponentåldern, desto äldre komponenter som används desto mer sannolikt är det att man använder sig av en förgången teknologi och att det kan föreligga högre sannolikhet att den ignoreras av säkerhetsforskare.

- Föråldrade komponenter kan vara en källa till komponenter som inte längre stöds av komponentutvecklare. När de inte längre stöds så blir de inte kontinuerligt uppdaterade.
- Kända sårbarheter i programvarukomponenter som beskrivs, till exempel i NVD (se nedan), är med säkerhet också kända av de som utnyttjar säkerhetsbrister. SCA brukar normalt inkludera flera källor för sårbarhetsinformation. Värt att notera är att även om man använder en komponent med en känd sårbarhet så betyder det inte alltid att sårbarheten per automatik finns i programmet.
- Komponenttyp, som ramverk och bibliotek har unika uppgraderingsutmaningar och associerad risk. Ett loggningsbibliotek är lättare att byta ut än ett webbapplikationsramverk, därför är det viktigt att utvärdera typen som en del av strategin för SCA.
- Komponentkvantitet, förutom att den potentiella risken kan reduceras om man minskar antalet komponenter, minskar kostnader för underhåll och utveckling.
- Förtroende för programvarupaket kan avgöras på om det finns krav på kodsignering och att verifiering har ett visst mått av förtroende.
- Ursprung är viktigt att ta i beaktande. Att veta att det är en betrodd skapare ökar sannolikheten att ingen främmande kod har smugit sig in.
- Programvarulicens är viktigt att ha koll på, då användandet av komponenter med licenser som är i konflikt med en organisations mål eller förmåga kan skapa allvarlig risk för organisationen.
- Externa tjänster är när applikationer, deras direkta eller indirekta komponenter förlitar sig på externa tjänster för funktionalitet. Det kan vara bekvämlighetsbibliotek runt vanliga webbtjänster såsom kartor, aktiekurser eller väder, men kan även inkludera autogenererade mjukvarukomponenter konstruerade från API-specifikationer som TUF eller OpenAPI.

Det är lämpligt att utforma sin analys utifrån riskfaktorerna ovan och ha det med som en del i sitt säkerhetsarbete.

Software Bill Of Material [Software supply chain]

Enkelt förklarat är det en mjukvaruförteckning för ett projekt, inkluderar alla komponenter där det finns ett beroende. Det finns två huvudstandarder för SBOMs:

- Software Package Data Exchange [SPDX] är en standard från the Linux Foundation.
- CycloneDX [CycloneDX] är en standard från OWASP Foundation.

Skillnaderna mellan dessa standarder är att SPDX fokuserar på licenser medans CycloneDX är fokuserat på säkerhetsaspekter. Även om standarderna har olika fokus så har de mycket gemensamt. De bygger båda på öppen källkod och syftar på att beskriva och hantera mjukvarukomponenter och deras beroenden. Båda kan användas vid licenshantering och säkerhetsarbete.

National Vulnerability Database [NVD]

National Vulnerability Database [NVD] är en centraliserad databas för CVE-listan som administreras av National Institute of Standards and Technology [NIST]. NVD beskriver tre typer av huvudkomponenter via CPE, dessa är applikationer (inkluderat bibliotek och frameworks), operativsystem och hårdvara. NVD analyserar även sårbarheterna och publicerar sina fynd tillsammans med CVE-rapporten. I rapporten inkluderas referenstaggar, Common Weakness Enumeration [CVE], Common Vulnerability Scoring System [CVSS] och Common Product Enumerator [CPE]. NVD innehåller mer än 248 000 CVE-rapporter om kända sårbarheter.

OWASP Dependency Check [ODC]

ODC är en programvara som gör en Software Composition Analysis [SCA] för att hitta kända sårbarheter i projektberoenden. Programmet börjar med att leta efter CPEer för givet bibliotek och om det hittas så kopplas det till associerade CVEer. Vid varje körning kontrolleras att NVD är lokalt uppdaterad. ODC producerar en rapport som kan fås i många format, bland annat HTML, XML, JSON och CSV. Man kan analysera projekt med .NET, Java, GoLang, npm och så vidare. ODC är en Java applikation som finns för många olika plattformar såsom Jenkins Plugin, Command Line, Maven Plugin, Gradle Plugin och Ant Task. En Docker-image finns också att tillgå. ODC är open source och är licensierat under Apache 2.0 licens.

Trivy

Trivy är ett program från Aqua Security som är open source [Trivy]. Med detta program har man möjlighet att skanna många olika typer av mål, som container images, filsystem, Git repositories, Virtual Machine Images, Kubernetes och AWS. Förutom att skanna efter kända sårbarheter (CVE), så kan det leta efter problem och felkonfigurationer i IaC (Infrastructure as Code), känslig information, hemligheter och mjukvarulicenser. Rapporter kan fås som bland annat JSON, Table, SBOM och SARIF. Man kan analysera projekt med .NET, Java, Node.js, Python och så vidare. Kan köras som ett program eller som en Docker container. Trivy är licensierat under Apache 2.0 licens.

Dependency Track

Dependency Track är en plattform för att kontinuerligt, bland annat övervaka programvarupaket och deras sårbarheter i mjukvaruprojekt [Dependency Track]. Dependency Track fungerar som så att den konsumerar SBOMs och kan sen utifrån dem kontinuerligt analysera dessa och se om det finns några nya sårbarheter eller förändringar i licenser. Den centrala delen av plattformen är servern Dependency-Track som kontinuerligt analyserar komponenter i ett eller flera projekt [Dependency Track Doc]. För att lagra data i produktion rekommenderas användningen av PostgreSQL eller Microsoft SQL Server och inte den inbyggda H2-databasen. För att hantera det hela enkelt, finns det en frontend som är lätt att komma åt från webbläsare. Det finns även möjlighet att skicka vidare analyserna till andra verktyg, såsom DefectDojo. Hela projektet är licensierat under Apache 2.0 licens.

CycloneDX CLI

Är ett verktyg från CycloneDX som bland annat kan generera, konvertera, jämföra, signera, verifiera SBOMs [CycloneDX]. Huvudorsaken till att jag tar upp denna klient är att den kan användas för att analysera ett projekt och generera en SBOM som sedan kan användas för att utföra SCA av projektet. Kan med fördel användas i en CI/CD miljö, där den kan generera en SBOM för varje version av projektet.

DefetctDojo

DefectDojo är ett verktyg med öppen källkod, för att underlätta hantering av data om sårbarheter och säkerhetstestning [DefectDojo]. Det har även funktioner för rapportering som man kan ha som hjälp vid analyser, för att nå insikter om sina sårbarheter. Så som att spåra vilka sårbarheter som finns i projekt över tid. Plattformen har stöd för integrationer med många olika verktyg och tjänster, vilket kan göra den till en central del i säkerhetsprocessen. Förutom den öppna källkodsvarianten så finns också Pro- och Enterprise-versionerna, som är kommersiella.

Andra verktyg för SCA

Det finns gott om olika verktyg för analys och hantering av SCA [Springett]. Vissa finns på specifika plattformar, andra har det som en tjänst (Software as a Service, SaaS). Licenserna varierar också från kommersiella till olika typer av öppna källkodslicenser. Jag har inte haft tid att gå igenom alla utan har varit tvungen att begränsa mina utforskningar inom området. Därför har jag valt att titta på ett fåtal verktyg med öppen källkodslicens och inte några kommersiella.

Sammanställning

För att kunna besvara frågeställning 1. “Hur kan man upptäcka sårbarheter i programvarupaket som påverkar ens egen kod?” så säger OWASP att man först måste göra en skanning av programvarupaketen som är inkluderade i projektet och sedan utföra en manuell analys av källkoden tillsammans med den information skanningverktyget har rapporterat. Ju mer specifik information som fås från skanningverktyget desto lättare blir den icke-automatiska analysen.

När man tar en titt på frågeställning 2 “Vad kan man göra för att skydda sig mot åldrande programvarupaket?” så finns det ett par sätt att kontinuerligt skydda sig. Den första metoden innebär att kontinuerligt köra ett verktyg, till exempel ODC, med jämna mellanrum med hjälp av en schemaläggare och ladda upp resultaten till ett program som används för att hantera potentiella sårbarheter. Det andra metoden är att ladda upp SBOMs till ett program som kontinuerligt analyserar informationen, som till exempel Dependency Track. Det är viktigt att ha uppdaterad information om kända sårbarheter i båda dessa metoder.

Kombinationen av möjliga lösningar är väldigt bred om man tittar på frågeställning 3 “Vilka verktyg finns tillgängliga för att övervaka nya sårbarheter och identifiera dem i den egna programvaran?”, men om man utgår från de studier som jag gjort så skulle man kunna ha två huvudspår likt svaret på frågeställning 2. För att köra schemalagd skanning i CI/CD så skulle man kunna använda till exempel Trivy eller ODC. Fördelen med Trivy gentemot ODC är att även IaC, känslig information, hemligheter och mjukvarulicenser kontrolleras. Rapporterna kan sedan laddas upp till ett program för hantering av potentiella sårbarheter så att vidare analys kan utföras och dokumenteras. För att köra kontinuerlig analys utanför CI/CD skulle kan kunna köra CycloneDX eller Trivy för att skapa SBOMs, som laddas upp till ett verktyg som kontinuerligt kontrollerar om det finns nyupptäckta sårbarheter i ett projekt som analyseras.

De tekniska aspekterna av frågeställning 4 “Hur kan SCA integreras i utvecklingsprocessen så att det blir en naturlig del av den?” besvaras till största delen av svaren till frågeställningarna 1 till 3. Men för att det skall bli en naturlig del av utvecklingsprocessen behövs att den skall finnas med i början på processen så att man tidigt kan identifiera sårbarheter. Att få in den i pipelinen och generera rätt feedback till rätt person. Att integrera SCA i IDEer kan hjälpa utvecklare tidigt i processen och med rätt kunskap kan brister i säkerheten minimeras. För att säkerställa funktionalitet även efter att produkten är "färdig" måste en kontinuerlig uppföljning finnas, eftersom nya säkerhetsbrister kan upptäckas i tidigare säkra programvarupaket.

Observation

Jag skriver här om mina erfarenheter under min LIA och är så faktabaserad som jag kan vara. Versionshanteringstjänsten som användes är GitLab Premium och projekten var alla hanterade av Maven, som är ett management verktyg för Java. Min första uppgift var att inkorporera Dependency Check Maven [ODC Maven] för ett antal projekt och att ladda upp rapporten till Pages och DefectDojo. Pages är ett snabbt sätt att skapa en webbsida för varje projekt i GitLab. Åtkomsträttigheter för sidorna är mycket restriktiva, vilket säkerställer att endast betrodd personal har enkel tillgång till information om programvarusårbarheter.

Alla programvarupaket som inkluderas i ett Maven projekt hanteras i en XML-fil som heter POM.xml och står för Project Object Model [POM]. För att det skall fungera lades pluginet ODC Maven till i POM-filen med konfigurationer som var tänkta att inte störa den normala processen i pipelinen mer än nödvändigt. Det betyder att i CI/CD pipelinen så var ODC Maven en integrerad del av byggprocessen för projekten och uppladdningen av rapporten till Pages och DefectDojo gjordes i separata steg. Programvarupaket som inkluderas i Maven sparas under en lokal katalog som heter .m2 och ODC Maven lagrar sin databas under den mappen så att den data inte behöver laddas ner konstant. Första gången som data från NVD hämtas kan det ta 20 minuter, men om man kör ODC minst en gång i veckan så laddas bara en liten uppdatering ner och det tar bara ett par sekunder [ODC].

Den konfiguration som fanns tillhanda för GitLab som jag arbetade mot, sparade NVD data i den för projektet lokala .m2 katalogen. Det gjorde att projekten behövde minst en commit i veckan, annars så tog byggprocessen 20 minuter extra tid att utföra. För utvecklare som kompilerade lokalt tog det 20 minuter första gången projektet byggdes, men eftersom projekten inte har separata .m2-kataloger utan har en gemensam katalog, så var det inte projektspecifikt. Eftersom projekten jag arbetade med var mogna, gjordes det inte alltid en commit varje vecka som laddades upp till GitLab. Detta medförde att en 20 minuters fördröjning av byggprocessen som normalt tar mindre än 5 minuter. Fördröjningen skapade irritation och var inte så populärt hos utvecklarna.

Under en period om tio dagar, i slutet av min första LIA-period, så hade NVD problem med att serva de som ville ladda ner NVD datan. Det gjorde att byggprocessen tog en timme, för det var den tiden det tog ODC Maven att göra timeout. För att hantera detta valde man antingen att kommentera bort ODC helt eller lägga till i konfigurationen att man inte skulle hämta de senaste

uppdateringarna från NVD. Om SCA var en källa till irritation hos utvecklarna innan, blev det än mer under denna period.

På grund av problemen som fanns så gjorde jag undersökningar på hur vi skulle kunna fortsätta använda ODC utan störningar. Tanken var att ODC skulle köra i ett separat steg i pipelinen och att NVD skulle lagras lokalt och uppdateras med automatik några gånger per dag. Jag hittade ett projekt under GitLab CI Utils som heter Docker Dependency Check [Goldenthal] som använder sig av ODCs Docker image och lägger till NVD datan i imagen. Pipelinen körs var fjärde timme och image:latest uppdateras med ny uppdaterad NVD data. Eftersom ODC nu inte är en integrerad del av byggprocessen utan istället utförs i ett separat steg som inte blockerar något annat steg, så blir störningarna för utvecklarna minimal med nästan samma funktionalitet. ODC körs nu inte längre automatiskt lokalt. Istället måste det antingen köras genom IDE:n eller installeras separat.

Även om SCA indikerar att ett specifikt programvarupaket är sårbart, behöver sårbarheten inte nödvändigtvis vara tillämplig på det aktuella projektet eftersom paketet kanske inte används på ett sätt som gör att sårbarheten kan utnyttjas. Om den möjliga sårbarheten inte är en sårbarhet kallas det att den är en falskt positivt [NIST False Positive]. För att kunna markera en möjlig sårbarhet som falskt positiv behövs en manuell undersökning av kodbasen av en utvecklare.

Med kunskapen om vilka programvarupaket som är sårbara i vilka projekt så fanns det nu möjlighet att kontrollera om uppdatering av programvarupaket och i vissa fall Java koden skulle minska möjliga sårbarheter.

En viktig del i uppgraderingsprocessen av programvarupaket är att verifiera att ingen funktionalitet har förändrats i projektet. För att kunna göra detta måste tester vara på plats och utföras. I de projekt där tester redan fanns på plats, testade jag att uppdatera programvarupaket och noterade att man enkelt kunde minska antalet sårbarheter utan att skriva om originalkoden, så länge man inte överskred större versionsuppgraderingar. Om man vill uppgradera till den absolut senaste versionen av vissa programvarubibliotek kan det dock kräva både en uppgradering av Javaversionen och en omarbetning av vissa grundläggande koncept och deras implementationer.

Sammanställning

För att besvara frågeställning 1, hur man upptäcker potentiella sårbarheter i ens kod, kan jag från mina observationer konstatera att det är effektivt att använda

verktyg med öppen källkod, såsom ODC, för att lista potentiella sårbarheter i projektet.

När det gäller att skydda sig mot åldrande programvarupaket, som nämns i frågeställning 2, anser jag att det är avgörande att regelbundet skanna projekten och ha möjlighet att följa upp potentiella sårbarheter.

För frågeställning 3, angående andra verktyg för SCA, kan jag säga att det finns alternativ. Även om jag huvudsakligen har erfarenhet av ODC, utförde jag enklare tester med CycloneDx och Dependency Track för att undersöka om de gav liknande resultat. Tyvärr var testerna inte tillräckligt djupgående för att dra slutsatsen att de är likvärdiga, men initialt såg det lovande ut.

När det gäller frågeställning 4, hur SCA kan bli en naturlig del av utvecklingsprocessen, visar mina observationer att det är viktigt att engagera utvecklarna genom hela processen. Det är också viktigt att förse dem med lämplig information för att inte överväldiga dem. Dessutom måste verktygen förbli pålitliga och inte skapa onödig börda. Att kunna markera falskt positiva sårbarheter är också en viktig aspekt för att undvika att sårbarheter upprepas i återkommande rapporter.

Diskussion

Resultatet av detta examensarbete visar på att det är viktigt att SCA implementeras på rätt sätt och att det är viktigt att få med sig utvecklarna på varje steg i utvecklingsprocessen.

Huvudpunkterna i min slutsats om hur man bäst arbetar med SCA är:

- Vara inbyggt i pipelinen och ske per automatik.
- Använda uppdaterad information om kända sårbarheter.
- Kontinuerlig skanning av projekt för att upptäcka om nya kända sårbarheter finns.
- Rapporterna skall vara tydliga, så att utvecklare enkelt vet vad sårbarheten gör, hur den fungerar och om det finns uppdaterade versioner som bör användas istället.
- SCA-verktygen skall vara enkla och funktionella att använda och ha möjlighet att markera en sårbarhet som falskt positiv för att uteslutas i efterföljande rapporter.

Andra slutsatser jag kan dra är att om man minimerar antalet möjliga sårbarheter genom att uppdatera till senare versioner av programvarupaket kan de manuella verifieringar utföras snabbare. Det betyder att projekten måste ha tester som kan verifiera att uppdateringar i programvarupaket inte påverkar programmets funktion. Om utvecklare överväldigas av en stor mängd möjliga sårbarheter, kan det minska deras motivation. Genom att erbjuda en tydlig och genomtänkt strategi för hur arbetsbördan kan minimeras ökar chanserna för framgång betydligt.

En säkerhetsaspekt som jag observerade när jag gick igenom mitt material som är värt att nämna, är att jag normalt inte förespråkar att använda image:latest av något som jag inte själv har byggt. Men i fallet med GitLab Utils “Docker Dependency Check” så är källan trovärdig, har varit igång sedan 2019 och därför kunde jag trots det motivera mig själv att använda denna lösning. En annan möjlig lösning hade varit att kopiera “Docker Dependency Check” för att ha mer kontroll på den.

Jag har funderat på fördelar och nackdelar med att implementera ett annat verktyg än OWASP Dependency-Check (ODC), som till exempel Trivy. Hur mycket effektivare skulle implementationen kunna vara om man även kunde utföra Secret Scanning och licenshantering samtidigt som man utför Software Composition Analysis, jämfört med att ha olika steg där specifika verktyg utför sin del av säkerhetsprocessen? Att integrera flera funktioner i ett enda verktyg, som Trivy

erbjuder, kan leda till betydande effektivitet. Med Trivy kan man inte bara utföra Software Composition Analysis, utan också Secret Scanning och licenshantering i ett enda arbetsflöde. Detta kan minska komplexiteten och förbättra hastigheten i säkerhetsgranskningen, eftersom man slipper byta mellan olika verktyg och arbetsflöden. Dessutom kan en enhetlig rapportering ge en mer sammanhängande översikt över säkerhetsstatusen och göra det enklare att prioritera åtgärder.

En nackdel med att samla alla funktioner i ett enda verktyg är risken för att "lägga alla ägg i en korg". Om verktyget skulle få problem eller visa sig ha begränsningar inom ett område, kan hela säkerhetsprocessen påverkas negativt. Dessutom kan specialiserade verktyg för enskilda uppgifter, såsom licenshantering eller Secret Scanning, erbjuda mer djupgående analyser och fler funktioner än ett multiverktyg. Därför kan det finnas fördelar med att använda flera specialiserade verktyg för att säkerställa den högsta möjliga nivån av noggrannhet och säkerhet i varje del av processen.

Min slutsats är att det viktigaste inte är hur man implementerar SCA, utan att alla delar i säkerhetsprocessen ingår för att skapa en heltäckande lösning och därmed uppnå en så säker kodbas som möjligt. En annan viktig aspekt är att det inte räcker att endast implementera de automatiska testerna, då hela säkerhetsprocessen, från utvecklares säkerhetskompetens till penetrationstester, är avgörande för att säkerställa att kodbasen är både säker och tillförlitlig.

Källförteckning

CNAs. “CVE Numbering Authorities (CNAs).” *CVE.org*, https://www.cve.org/ProgramOrganization/CNAs. Accessed 6 May 2024.

CPE. “NVD - CPE.” *NVD*, https://nvd.nist.gov/products/cpe. Accessed 6 May 2024.

CVE. “CVE Program Overview.” *CVE.org*, https://www.cve.org/About/Overview. Accessed 6 May 2024.

CVSS. “Common Vulnerability Scoring System.” *Wikipedia*, https://en.wikipedia.org/wiki/Common_Vulnerability_Scoring_System. Accessed 6 May 2024.

CycloneDX. “Home.” *OWASP CycloneDX Software Bill of Materials (SBOM) Standard*, https://cyclonedx.org/. Accessed 8 May 2024.

CycloneDX-CLI. “CycloneDX/cyclonedx-cli: CycloneDX CLI tool for SBOM analysis, merging, diffs and format conversions.” *GitHub*, https://github.com/CycloneDX/cyclonedx-cli. Accessed 8 May 2024.

DefectDojo. “Home.” *DefectDojo | CI/CD and DevSecOps Automation*, https://www.defectdojo.org/. Accessed 9 May 2024.

Dependency Track. “Home.” *Dependency-Track | Software Bill of Materials (SBOM) Analysis*, https://dependencytrack.org/. Accessed 7 May 2024.

Dependency Track Doc. *Introduction*, https://docs.dependencytrack.org/. Accessed 7 May 2024.

Figur 1. “OWASP Top Ten.” *OWASP*, https://owasp.org/www-project-top-ten/assets/images/mapping.png. Accessed 13 May 2024.

Figur 2. “OWASP DevSecOps Guideline - v-0.2.” *OWASP*, https://owasp.org/www-project-devsecops-guideline/latest/assets/images/Pipeline-view.png. Accessed 13 May 2024.

Goldenthal, Aaron. “Docker dependency check.” *GitLab CI Utils*, https://gitlab.com/gitlab-ci-utils/docker-dependency-check/. Accessed 9 May 2024.

Layola Marymount University. “RADAR Framework - Evaluating Sources: Using the RADAR Framework - LibGuides at Loyola Marymount University.” *LibGuides*, 11 January 2024, https://libguides.lmu.edu/aboutRADAR. Accessed 6 May 2024.

Mandalios, Jane. “RADAR: An approach for helping students evaluate Internet sources.” *Sage Journal*, 6 January 2013, https://journals.sagepub.com/doi/abs/10.1177/0165551513478889. Accessed 6 May 2024.

MITRE. “Our Story.” *MITRE*, https://www.mitre.org/who-we-are/our-story. Accessed 5 May 2024.

MITRE CVE. “History.” *CVE.org*, https://www.cve.org/About/History. Accessed 6 May 2024.

NIST. “NVD - CVEs and the NVD Process.” *NVD*, https://nvd.nist.gov/general/cve-process. Accessed 5 May 2024.

NIST False Positive. “False Positive - Glossary | CSRC.” *NIST Computer Security Resource Center*, https://csrc.nist.gov/glossary/term/false_positive. Accessed 11 May 2024.

NVD. “National Vulnerability Database.” *NVD*, http://nvd.nist.gov. Accessed 2 May 2024.

ODC. “jeremylong/DependencyCheck: OWASP dependency-check is a software composition analysis utility that detects publicly disclosed vulnerabilities in application dependencies.” *GitHub*, https://github.com/jeremylong/DependencyCheck. Accessed 5 May 2024.

ODC Maven. *dependency-check-maven – Usage*, https://jeremylong.github.io/DependencyCheck/dependency-check-maven/. Accessed 9 May 2024.

OWASP. *OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation*, http://owasp.org. Accessed 2 May 2024.

POM. “Maven – POM Reference.” *Apache Maven*, https://maven.apache.org/pom.html. Accessed 9 May 2024.

SCA. “Software composition analysis.” *Wikipedia*, https://en.wikipedia.org/wiki/Software_composition_analysis. Accessed 6 May 2024.

Software supply chain. “Software supply chain.” *Wikipedia*, https://en.wikipedia.org/wiki/Software_supply_chain. Accessed 8 May 2024.

SPDX. “Overview – SPDX.” *SPDX*, https://spdx.dev/about/overview/. Accessed 8 May 2024.

Springett, Steve. “Component Analysis.” *OWASP Foundation*, https://owasp.org/www-community/Component_Analysis. Accessed 7 May 2024.

Trivy. “aquasecurity/trivy: Find vulnerabilities, misconfigurations, secrets, SBOM in containers, Kubernetes, code repositories, clouds and more.” *GitHub*, https://github.com/aquasecurity/trivy. Accessed 6 May 2024.