ZOSS – opis projekta

Uvod

DevSecOps predstavlja nadogradnju na DevOps ciklus razvoja softvera, koja promoviše *shift-security-left*, *security-by-design* i *continuous security testing*, tako što će se praviti automatizovane sigurnosne kontrole u DevOps *workflow*-u. Tokom isporuke projekta u produkciono okruženje, neophodno je izvršiti priličan broj provera nad samim softverskim rešenjem. Ove provere podrazumevaju testiranje (automatsko i manuelno) proizvoda, kako bi se osigurao ispravan i očekivan rad samog softvera koji se razvija, međutim neophodno je izvršiti i provere koje se tiču bezbednosti. Kako je ovaj proces veoma repetativan, zamoran za ljude a treba ga izvršavati što češće, on se može veoma lako automatizovati.

Jedan od izazova koji se tiče bezbednosnih praksi su teškoće u ostvarivanju potpune automatizacije tradicionalnih bezbednosnih praksi, koje se inače obavljaju ručno. Tu spadaju dizajn privatnosti, analiza arhitekturalnih rizika, modelovanje pretnji i upravljanje rizicima. U okviru DevSecOps-a je teško sprovesti brze procene bezbednosnih zahteva u sklopu brzog CD-a. U DevOps okruženju je zbog stalnih i brzih izlazaka verzija softvera teško izmeriti bezbednost softvera.

Važne komponente *DevSecOps*-a su: Modelovanje pretnji i procena rizika, Kontinualno testiranje, Monitoring i logovanje, *Security as code*, *Red-Team* i *security drills*, Ljudi, Alati, Prakse i Infrastruktura. Fokus ovog istraživanja je je na komponenti Alati.

Prilikom bezbednosnih provera postoji priličan broj provera koje je moguće izvršiti, a fokus ovog istraživanja je pregled 4 najbitnije vrste provera, alata koji su dostupni za ove vrste i praktična demonstracija u vidu implementacije pipeline-a i prikaza rada gde se ne detektuje mana (zato što je stvarno nema) i kada se detektuje mana (zato što je stvarno ima).

4 najbitnije vrste bezbednosnih provera pomoću alata su:

1. Provera sadržavanja *Git Secrets*-a – da li sam source kod sadrži neku tajnu (*secret*)
2. Provera *Source code analysis* – provera ranjivosti u *3rd party* bibliotekama koje softver koji razvijamo koristi
3. *Static Application Security Testing* (SAST) - analiza *source* koda rešenja koje razvijamo kako bi se pronašle ranjivosti
4. *Dynamic Application Security Testing* (DAST) - simuliranje ponasanja napadaca, izvodi se na serveru koji se koristi za testiranje (nakon *deploy*-ovanja aplikacije)

Ono što neće biti fokus ovog istraživanja jeste razvijanje same aplikacije, već će se koristiti bilo koja *Java* aplikacija. Takođe uprkos postojanju mnoštva build alata, oni se neće obrađivati u ovom radu, već će se koristiti *Maven build* alat. Razvoj i demonstracija pipeline-a će se obaviti u lokalu (odnosno ručnim pokretanjem na jednom računaru), pravi pipeline-ovi se hostuju na serverima (serveri mogu biti u cloud okruženju ali i on-prem). Pokretanje automatskih testova i izvršavanje manuelnih testova kako bi se

proverio rad samog softverskog rešenja koje se razvija nije fokus ovog istraživanja i neće biti implementirano u *pipeline*-u.

Nakon završetka rada ova 4 alata izveštaji koji su alati generisani će se *upload*-ovati u alat *Defect Dojo* koji prikuplja izveštaje i koristi se za *vulnerability management*.

Git Secrets

Secrets (tajne) se danas koriste svuda (API ključevi, sertifikati, kredencijali baze podataka, SSH ključevi, lozinke, itd). Često se dešava da organizacije ostave ove osetljive podatke u *plain-text* formatu u konfiguracionim fajlovima ili u samom kodu projekta.

Iz ovog razloga postoje različiti koncepti najboljih praksi kojih programeri i developeri treba da se pridržavaju, kao i alati koji tome pomažu. Neke od tih praksi su: dodavanje osetljivih fajlova u .gitignore, konstantno menjanje tajni (tj, korišćenje tajni sa kratkim životnim vekom), *whitelisting* IP adresa (ako je to moguće), definisanje minimalnih potrebnih privilegija (*permissions*) API-jima koji su u upotrebi, itd.

Jedna od najbitnijih praksi jeste automatizacija pronalaženja tajni u kodu. Iako se za slične potrebe (tj. proveru koda) koristi *code-review*, on nije pouzdan jer je čovek taj koji proverava kod, a čovek je sklon greškama. U te svrhe se koriste alati koji mogu taj posao uraditi umesto čoveka. Često se ovi alati mogu i integrisati u CI/CD *pipeline*

Alati koji služe za automatizaciju pronalaženja tajni (i ostalog) su: GitGuardian (*ggshield* - CLI API za GitGuardian), GitLeaks, Trufflehog, GitRob, GitWatchman, itd.

SCA

Software Composition Analysis (SCA) predstavlja automatizovani proces koji se bavi analizom otvorenog softvera. Preciznije, SCA se bavi evaluacijom bezbednosti, usklađenosti licenci i kvaliteta koda. Većina današnjih aplikacija je napravljena i pravi se pomoću otvorenog softvera. Korišćenjem otvorenog softvera ubrzavamo i olakšavamo proces kreiranja aplikacija, smanjujemo neophodne troškove i kreiramo bolji i kvalitetniji softver. Problemi ovog pristupa nastaju kada posmatramo bezbednost.

Softver otvorenog tipa nije bezbedan po difoltu i može da ima ranjivosti. Postoje dve vrste ranjivosti: one opasnije, "Unknown vulnerabilities" odnosno nepoznate ranjivosti, i one manje opasne, "Known vulnerabilities". Kada se otkrije nepoznata ranjivost, ona se šalje u CVE (*Common Vulnerabilities and Exposures*) bazu podataka. Primer jedne takve CVE baze podataka jeste NVD, koja predstavlja bazu podataka o upravljanju ranjivostima u SAD-u. Nakon što nepoznata ranjivost završi u CVE bazi, postaje poznata ranjivost. CVE daju skor od 1 do 10 na osnovu lakoće eksploatacije. SCA koristi CVE baze podataka da bi naše aplikacije sačuvale pri kreiranju i korišćenju.

U sistemu koji koristi softver otvorenog koda, imamo sledeću strukturu:

- Paket otvorenog koda (*open-source package*): datoteke i metapodaci koji opisuju izvorni kod projekta
- Menadžer paketa (*Package manager*): program koji primenjuje, automatizuje i upravlja paketima
- Registar paketa (*Package Registry*): usluga hostinga koja pruža mogućnost uvoza paketa
- Zavisnost (*Dependency*): softver koji koristi pakete ili biva korišćen kao paket

Ako se naš projekat oslanja na paket otvorenog koda, onda naš projekat zavisi od njega, što predstavlja direktnu zavisnost. Ukoliko naš projekat koristi paket otvorenog koda, a taj paket otvorenog koda koristi neki drugi paket otvorenog koda, dolazimo do indirektne zavisnosti.

Ručno praćenje svih ovih obaveza je vremenom postalo previše. Često se previđao kod od njegovih pratećih ranjivosti. Iz toga je i nastala potreba za automatizacijom tog sistema. Ranije i kontinuirano SCA testiranje omogućilo je programerima i bezbednosnim timovima da povećaju produktivnost bez ugrožavanja bezbednosti i kvaliteta softvera.

SCA automatski prati prethodno spomenute obaveze i sve otvorene softvere koje identifikuje ubacuje u BOM (*Bill of Materials*), koji se zatim upoređuje sa raznim bazama poznatih ranjivosti.

SAST

Statičko testiranje bezbednosti aplikacija (SAST) se koristi za obezbeđivanje softvera pregledom izvornog koda softvera da bi se identifikovali izvori ranjivosti. SAST alati se fokusiraju na sadržaj programskog koda aplikacije i njegove komponente kako bi utvrdili potencijalne bezbednosne propuste, *white-box* testiranje. U životnom ciklusu razvoja softvera (SDLC), SAST je poželjno izvoditi u ranoj fazi. Iako mnogi testovi rezultuju lažno pozitivnim (*false-positive*) rezultatima i ometaju usvajanje od strane programera, dobra praksa je koristiti ih.

SAST alati skeniraju kompletan izvorni kod (source code) i pokrivenost može da bude čak i 100%, dok dinamičko testiranje DAST pokriva njegovo izvršavanje, koje ne mora biti 100%-tno i takođe ne testira neobezbeđenu konfiguraciju u konfiguracionim fajlovima.

SAST alati mogu ponuditi i proširene funkcionalnosti kao što su testiranje kvaliteta i testiranje arhitekture rešenja. Između kvaliteta softvera i sigurnosti postoji direktna korelacija, jer softver lošijeg kvaliteta je vrlo verovatno i loše obezbeđen softver.

Iako programeri žele da koriste SAST alate, postoje izazovi koje programeri trebaju da prebrode. Upotrebljivost izlaza (izveštaja) koji se generiše od strane alata može uticati na to koliko programeri mogu da iskoriste ove alate. Rana upotreba SAST alata prilikom rada po Agilnim metodologijama može proizvesti dosta bagova, jer programeri koji koriste ovu metodologiju se fokusiraju najviše na funkcionalnosti i isporuku rešenja. Alati mogu generisati mnogo lažno pozitivnih (false-positive)

problema i time povećaju vreme neophodno za istraživanje problema i smanjuje se poverenje programera u sam alat.

Alati koji bi se istraživali su: SonarQube, Veracode, Checkmarx, Fortify, Coverity. Nakon istraživanja alata jedan bi se iskoristio za implementaciju u samom pipeline-u.

DAST

DAST predstavlja testiranje u okviru kog se simulira ponašanje napadača, odnosno veb aplikacija se napada spolja. Izvršava se nakon *deploy*-a aplikacije. Nezavisno je od programskog jezika i korišćenih tehnologija.

Ranjivosti koje se najčešće otkrivaju na ovaj način su: XSS, SQL *injection*, CSRF, *Denial of service*, *Arbitrary code execution*, *Memory corruption*, *Information disclosure*.

Posmatraju se dve tehnike:

1. *Web Application Security Testing* (WAST)
2. *Security API Scanning* (SAS)

WAST je tehnika u okviru koje se aplikacija napada preko korisničkog interfejsa. Izvršava se u 3 koraka:

1. *Spider scan* – vrši se *crawling* veb aplikacije, da bi se otkrili svi dostupni URL-ovi i resursi, odnosno traži se površina napada veb aplikacije i pravi se reprezentacija strukture aplikacije.
2. *Active scan* – šalju se maliciozni zahtevi za svaki identifikovani resurs i analiziraju se dobijeni odgovori kako bi se utvrdilo da li postoje ranjivosti.
3. *Result reporting* – dobijeni rezultati se agregiraju u izveštaj. Izveštaj treba ručno da se analizira da bi se odbacili *false positive*-i.

SAS je tehnika u okviru koje se veb servis napada preko API-ja. Na ovaj način je moguće detaljno testirati svaki *endpoint*. Glavni cilj je da se identifikuju ranjivosti vezane za autentifikaciju, autorizaciju, validaciju *input*-a, *error handling* i sl. Zahtev se API-ju šalje pomoću *request* komponente. Ona omogućava kreiranje i slanje HTTP zahteva ciljnom API-ju. Svi zahtevi prolaze kroz *proxy* komponentu koja presreće saobraćaj između *request* komponente i aplikacije. Ona omogućava inspekciju i izmjenu HTTP zahtjeva i odgovora. Ona evaluira presretnuti saobraćaj radi pronalaska ranjivosti. Nakon obavljenog testa, *proxy* komponenta kreira izveštaj sa rezultatima evaluacije.

Alati za DAST su OWASP ZAP i *Burp Suite*.

Literatura:

- Myrbakken, H. and Colomo-Palacios, R., 2017. DevSecOps: a multivocal literature review. In *Software Process Improvement and Capability Determination: 17th International Conference, SPICE 2017, Palma de Mallorca, Spain, October 4–5, 2017, Proceedings* (pp. 17-29). Springer International Publishing.
- Rajapakse, R.N., Zahedi, M., Babar, M.A. and Shen, H., 2022. Challenges and solutions when adopting DevSecOps: A systematic review. *Information and software technology*, *141*, p.106700.
- Zaydi, M. and Nassereddine, B., 2020. DevSecOps PRACTICES FOR AN AGILE AND SECURE IT SERVICE MANAGEMENT. *Journal of Management Information & Decision Sciences*, *23*(2).
- Prates, L., Faustino, J., Silva, M. and Pereira, R., 2019. Devsecops metrics. In *Information Systems: Research, Development, Applications, Education: 12th SIGSAND/PLAIS EuroSymposium 2019, Gdansk, Poland, September 19, 2019, Proceedings 12* (pp. 77-90). Springer International Publishing.