

TEHNIČKO VELEUČILIŠTE U ZAGREBU

POLITEHNIČKI SPECIJALISTIČKI DIPLOMSKI STRUČNI STUDIJ

Specijalizacija informacijska sigurnost i digitalna forenzika

Dragan Bošnjak

**IMPLEMENTACIJA SIGURNOSNIH ASPEKATA
U ŽIVOTNOM CIKLUSU RAZVOJA SOFTVERA**

DIPLOMSKI RAD br. IS 25

Zagreb, rujan 2020.

TEHNIČKO VELEUČILIŠTE U ZAGREBU

POLITEHNIČKI SPECIJALISTIČKI DIPLOMSKI STRUČNI STUDIJ

Specijalizacija informacijska sigurnost i digitalna forenzika

IMPLEMENTACIJA SIGURNOSNIH ASPEKATA U ŽIVOTNOM CIKLUSU RAZVOJA SOFTVERA

DIPLOMSKI RAD br. IS 25

Povjerenstvo:

mr.sc., Marinko, Žagar, predsjednik povjerenstva

dr.sc., Damir, Delija, član

dipl.ing., Aleksander, Radovan, mentor

Zagreb, rujan 2020.

Sažetak

Rad govori o poboljšanju sigurnosti softvera iz perspektive razvojnog tima. Opisane su aktivnosti koje se mogu koristiti s ciljem proizvodnje sigurnijeg i pouzdanijeg softvera, prvenstveno smanjivanjem broja stvorenih ranjivosti te omogućavanjem ranog otkrivanja ranjivosti. Aktivnosti pokrivaju sve faze životnog ciklusa razvoja softvera, od definiranja zahtjeva, dizajna, implementacije, testiranja pa sve do primjene.

Navedene su najčešće korištene metodologije i radni okviri koji pomažu unapređenju sigurnosti u procesu razvoja softvera. Opisane su njihove značajke i način primjene u organizacijama.

Na kraju je opisan primjer implementacije procesa kontinuirane integracije u kojemu su pomoću automatiziranih alata izvedene neke od u radu navedenih sigurnosnih aktivnosti za prevenciju i otkrivanje ranjivosti. Primjer pokazuje kako na učinkovit način unaprijediti razvoj softvera bez potrebe za većim ulaganjem i znatnim promjenama ustaljenog razvojnog procesa.

Ključne riječi: sigurnost, softver, proces, razvoj, testiranje.

Sadržaj

1.	Uvod	1
2.	Životni ciklus razvoja softvera.....	3
2.1.	Procesi i modeli.....	3
2.2.	Generalizirani model	3
2.3.	Siguran proces razvoja softvera.....	4
3.	Utvrđivanje zahtjeva	6
3.1.	Prikupljanje sigurnosnih zahtjeva.....	6
3.2.	Utvrđivanje prihvatljivih granica kvalitete	7
3.3.	Procjena rizika	10
4.	Analiza i dizajn	11
4.1.	Primjena sigurnog dizajna.....	11
4.2.	Pregled arhitekture i dizajna	13
4.3.	Modeliranje prijetnji	14
5.	Implementacija	18
5.1.	Primjena praksi sigurnog kodiranja	18
5.2.	Pisanje testova.....	20
5.3.	Pregled koda.....	21
5.4.	Statička analiza koda	23
6.	Verifikacija i validacija.....	26
6.1.	Sigurnosno testiranje	26
6.2.	Skeniranje ranjivosti.....	27
6.3.	Penetracijsko testiranje	27
6.4.	Testiranje Fuzz metodom	29
6.5.	Osiguravanje lanca dobavljača softvera	29
7.	Primjena	32
7.1.	Pregled produkcije	32
7.2.	Nadziranje i obavještanje	33
7.3.	Odgovori na incidente	35
7.4.	Post mortem.....	36
8.	Metodologije i radni okviri	38

8.1.	Microsoft Security Development Lifecycle (SDL)	38
8.2.	OWASP SAMM	41
8.3.	BSIMM	43
9.	Kontinuirana integracija	47
10.	Zaključak	55
	Literatura	56

Popis oznaka i kratica

ACL	Access-control list
ASVS	Application Security Verification Standard
BSIMM	Building Security In Maturity Model
CI	Continuous Integration
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DAST	Dynamic Application Security Testing
HIDS	Host-based intrusion detection system
IAST	Interactive Application Security Testing
IDE	Integrated development environment
KISS	Keep it simple, stupid
MAC	Mandatory access control
NIST	National Institute of Standards and Technology
OWASP	Open Web Application Security Project
PCI DSS	Payment Card Industry Data Security Standard
RASP	Run-time Application Security Protection
SAAS	Software as a service
SAMM	Software Assurance Maturity Model
SAST	Static Application Security Testing
SCA	Software Composition Analysis
SDL	Microsoft Security Development Lifecycle
SDLC	Software Development Life Cycle
SIEM	Security information and event management
SLA	Service-level agreement

Popis tablica

Tablica 1: Relativni trošak popravka neispravnosti pronađenih u različitim fazama razvoja softvera	2
Tablica 2: Akcije u fazama razvoja softvera.....	5

Popis slika

Slika 1: Faze modela procesa razvoja softvera	4
Slika 2: Dijagram toka podataka u alatu Microsoft Threat Modeling Tool	15
Slika 3: Upozorenja alata za statičku analizu koda unutar IDE sučelja.....	24
Slika 4: The Security Development Lifecycle (SDL)	39
Slika 5: Rezultati procjene BSIMM prikazani su u obliku grafikona	45
Slika 6: Oznake rezultata provjera procesa kontinuirane integracije	48
Slika 7: Prikaz rezultata provjera promjena koda.....	48
Slika 8: Cjevovod kontinuirane integracije	49
Slika 9: Sigurnosni propusti detektirani korištenjem alata Bandit.....	49
Slika 10: Izvještaj o pokrivenosti koda	50
Slika 11: Komentar servisa LGTM na stranici za pregled promjena koda	50
Slika 12: Propusti detektirani servisom LGTM	51
Slika 13: Prikaz rezultata skeniranja servisom SonarCloud.....	52
Slika 14: Postavke granica kvalitete u servisu SonarCloud	52
Slika 15: Komentar servisa SonarCloud na stranici za pregled promjena koda	53
Slika 16: Rezultat analize koda servisom Code Climate Quality.....	53
Slika 17: Prikaz detalja o ranjivosti ovisnosti otkrivene Snykom	54

1. Uvod

Softver je posvuda. S razvojem tehnologije i inovacijama, širi se prisutnost softvera. Ljudi danas često koriste softver, a da toga nisu niti svjesni. Softver je ključan dio tehnologija koje poboljšavaju živote ljudi i unapređuju poslovanje različitih industrija. S porastom prisutnosti softvera i tehnologije, te ovisnosti društva o njima, rastu i mogućnost i razlozi za njihovu zloupotrebu. Drugim riječima, raste rizik.

Kibernetički kriminal zlouporabom tehnologije globalnom gospodarstvu nanosi štetu koja se procjenjuje na oko 600 milijardi dolara godišnje [1]. Tvrtke ulažu milijune dolara u sigurnosne programe kako bi zaštitile kritične infrastrukture, identificirale sigurnosne probleme i spriječile maliciozne napade i curenje podataka. Očito postoji potreba da softver bude siguran i pouzdan.

Ranjivosti koje narušavaju sigurnost softvera se mogu pojaviti u obliku nedostataka (engl. *defect*) u implementaciji i dizajnu softvera. Kada postojanje ovih nedostataka narušava sigurnost sustava, riječ je o ranjivostima. Ranjivosti mogu biti prisutne u softveru godinama bez posljedica za djelovanje sustava, no njihovo otkrivanje i iskorištavanje može dovesti do ogromnih posljedica [2, str. 14].

Nedostatci koji nastaju u implementaciji softvera se nazivaju pogreške ili *bugovi*. Radi se o pogreškama na niskoj razini implementacije, to jest u samom kodu, čiji je rezultat neželjeno ponašanje sustava. Jednom kada se pogreške otkriju, često ih je lako popraviti. Primjer pogrešaka koji mogu biti uzrok ranjivosti su preljev spremnika (engl. *buffer overflow*), preljev cijelog broja (engl. *integer overflow*), neispravna validacija ulaznih podataka, stanje trke (engl. *race condition*), itd.

Nedostatci u dizajnu softvera se nazivaju mane. Mane su puno dublji problemi od jednostavnih *bugova*. U ovom slučaju sustav se ponaša kako je zamišljen, ali problem je inherentan zbog samog dizajna [2, str. 16], [3, str. 47]. Primjeri su nepostojanje autentikacije ili loši autentikacijski mehanizmi, korištenje nesigurnih protokola, nesigurno postupanje s osjetljivim podacima, neučinkovit ili nesiguran sustav za oporavak od pogrešaka itd. Ispravljanje mana u dizajnu može zahtijevati značajnije promjene i restrukturiranje aplikacije, što stvara nove rizike od *bugova* i ranjivosti.

Bitno je primijetiti da ranjivosti mogu nastati u bilo kojoj fazi razvoja softvera, na primjer [4, str. 5]:

- Tokom planiranja se ne definiraju sigurnosni zahtjevi.
- Dizajn i arhitektura sadrže sigurnosne propuste.
- Korištenje loših praksi kodiranja prilikom implementacije.
- Propusti prilikom puštanja u primjenu.

Prema izvještaju NIST-a, trošak popravka neispravnosti kroz faze razvoja softvera raste eksponencijalno: trošak popravka neispravnosti pronađenih u post-produkcijskoj fazi razvoja softvera je i do 30 puta veći u odnosu na trošak popravka neispravnosti pronađenih u fazi planiranja i dizajniranja [5, str. 94].

Tablica 1: Relativni trošak popravka neispravnosti pronađenih u različitim fazama razvoja softvera

Prikupljanje i analiza zahtjeva / Dizajn arhitekture	Implementacija	Integracijsko testiranje	Rana povratna Informacija korisnika / beta test	Post-produkcijsko izdanje
1X	5X	10X	15X	30X

Zbog ovoga je smanjivanje broja nedostataka otkrivanjem u što ranijim fazama razvoja učinkovita strategija za smanjivanje troškova. Ako se uz ovo uzmu u obzir potencijalni troškovi koji mogu nastati kod iskorištavanja sigurnosnih propusta, postaje očito kako je razvoj softvera koji je siguran i pouzdan puno jeftiniji od ispravljanja nedostataka poslije razvoja [4, str. 3].

Možda je nemoguće potpuno izbjeći stvaranje ranjivosti i proizvesti potpuno siguran softver, ali moguće je unaprijediti sigurnost smanjivanjem broj ranjivosti koje nastaju u softveru, te otkrivanjem nastalih ranjivosti kako bi se ispravile na vrijeme i izbjegla potencijalna šteta. Pitanje je kako to izvesti na što učinkovitiji način [6, str. 4].

2. Životni ciklus razvoja softvera

2.1. Procesi i modeli

Proces razvoja softvera ili životni ciklus razvoja softvera (engl. *software development life cycle, SDLC*) je skup aktivnosti koje se provode s ciljem proizvodnje softvera.

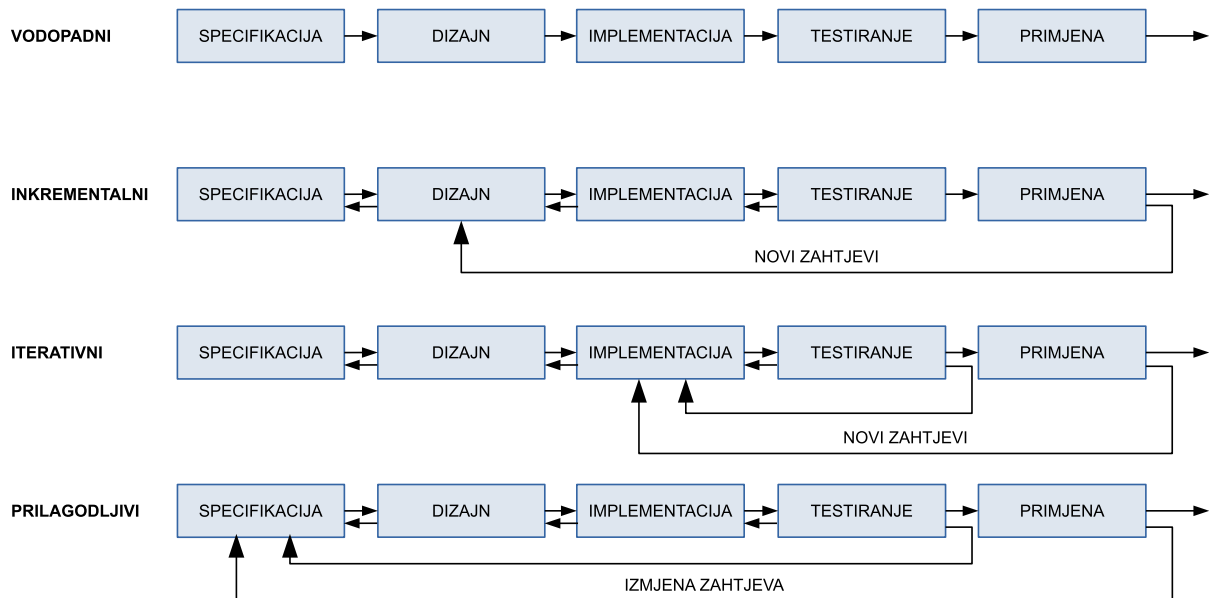
Postoji više modela procesa razvoja softvera. Neki od modela su [7, str. 35]:

- Vodopadni model (engl. *waterfall*): Linearni proces koji je primjenjiv kada su korisnički zahtjevi dobro definirani. Aktivnosti se odvijaju kroz sekvencijalne faze bez povratne informacije. Proces završava završetkom posljednje faze.
- Inkrementalni (evolucijski) model: Koristi se kada projekt mora isporučiti rezultat ranije u životnom ciklusu softvera. Aktivnosti se unutar iteracije odvijaju sekvencijalno, a rezultat svake iteracije je djelomično rješenje.
- Iterativni model: Faze procesa su povezane u grupe povratnom vezom. Nakon izvršenja svake faze nastavak izvršavanja ovisi o povratnoj informaciji. Posljednja faza ovisno i povratnoj informaciji rezultira djelomičnim rješenjem.
- Prilagodljivi model: Primjenjuje se kod potrebe čestih promjena zahtjeva (specifikacija). Aktivnosti se izvršavaju u ciklusima, a nakon završetka svakog ciklusa slijedi ponovno utvrđivanje zahtjeva.

Svaki model ima prednosti i mane i primjenjiv je u odgovarajućim situacijama, pa ih organizacije usvajaju i prilagođavaju ovisno o svojim potrebama i mogućnostima.

2.2. Generalizirani model

Procese razvoja softvera je moguće generalizirati promatranjem njihovih sličnosti i zajedničkih karakteristika.



Slika 1: Faze modela procesa razvoja softvera [8, str. 59]

Generalno, modeli procesa dijele sljedeće osnovne faze:

- utvrđivanje zahtjeva (specifikacija)
- analiza i dizajn
- implementacija (programiranje)
- verifikacija i validacija (testiranje)
- primjena (produkcija).

2.3. Siguran proces razvoja softvera

Neovisno o modelu, proces razvoja softvera se može poboljšati tako da se u svaku fazu generaliziranog modela uključe specifične sigurnosne aktivnosti s ciljem sprječavanja nastanka i pravovremenog otkrivanja propusta. Ovakav proces se naziva sigurnim procesom razvoja softvera [6, str. 6], [9, str. 4], [10, str. 11].

Tablica 2: Akcije u fazama razvoja softvera

Utvrđivanje zahtjeva	Analiza i dizajn	Implementacija	Verifikacija i validacija	Primjena
Prikupljanje sigurnosnih zahtjeva	Primjena sigurnog dizajna	Primjena praksi sigurnog kodiranja	Sigurnosno testiranje	Pregled produkcije
Utvrđivanje prihvatljivih granica kvalitete	Pregled arhitekture i dizajna	Pisanje testova	Skeniranje ranjivosti	Nadziranje i obavješćavanje
Procjena rizika	Modeliranje prijetnji	Pregled koda	Penetracijsko testiranje	Odgovori na incidente
		Statička analiza koda	Testiranje fuzz metodom	Post mortem
			Osiguravanje lanca dobavljača softvera	

3. Utvrđivanje zahtjeva

3.1. Prikupljanje sigurnosnih zahtjeva

Prepoznavanje sigurnosnih zahtjeva i definiranje djelotvornih sigurnosnih kontrola ključan je aspekt učinkovitog sigurnog procesa razvoja softvera. Bez obzira na razvojnu metodologiju, definiranje sigurnosnih kontrola aplikacije započinje u fazi planiranja i nastavlja se tijekom životnog ciklusa aplikacije [11, str. 7].

Dokumentacija sigurnosnih zahtjeva počinje s razumijevanjem funkcionalnih zahtjeva. Funkcionalni zahtjevi pružaju informacije o očekivanoj funkcionalnosti aplikacije. Sigurnosni zahtjevi naglašavaju potrebu o zaštiti podataka i privatnosti korisnika, usklađenosti s propisima i praćenju standarada [12, str. 21].

Pregledom funkcionalnih zahtjeva identificiraju se odgovarajuća očekivanja s obzirom na potrebe o povjerljivosti, integritetu i dostupnosti podataka ili usluga koji su dio funkcionalnih zahtjeva. Zahtjevi moraju navoditi cilj (npr. „Podaci se moraju prenositi sigurnim kanalom“) ali ne nužno i točnu mjeru za postizanje cilja (npr. „Podaci se moraju prenositi TLSv1.2 protokolom“) [13, str. 28].

Sigurnosni zahtjevi mogu biti funkcionalni ili nefunkcionalni. Funkcionalni sigurnosni zahtjevi opisuju funkcionalno ponašanje koje provodi sigurnost (npr. „Svi ulazni podaci moraju biti validirani“). Nefunkcionalni zahtjevi opisuju kakav sustav mora biti (npr. „API mora biti otporan na SQL injekcije“) [14].

Sigurnosni zahtjevi se pišu slično funkcionalnim zahtjevima. Moraju biti jasni, nedvosmisleni i dosljedni, a ciljevi zahtjeva ispitivi i mjerljivi. Definiranjem specifičnih i ostvarivih zahtjeva, arhitekti i programeri mogu ispuniti sigurnosne ciljeve za aplikaciju [14].

U agilnom razvojnom procesu, zahtjevi se pišu u obliku korisničkih priča, to jest očekivanih scenarija iz korisničke perspektive. Na primjer: „Kao korisnik foruma, želim moći promijeniti sadržaj svog posta.“ Uz korisničku priču navode se kriteriji prihvatljivosti. Njima se definiraju kriteriji koje implementacija mora zadovoljiti da bi zahtjev bio ispunjen. Primjer sigurnosnih kriterija prihvatljivosti za ovaj slučaj:

- Korisnik mora biti autenticiran.

- Korisnik ne smije moći promijeniti sadržaj tuđeg posta.
- Svaka promjena mora biti logirana.

OWASP Application Security Verification Standard (ASVS) je popis standardnih sigurnosnih zahtjeva i kontrola sigurnosti aplikacija koje se koriste pri planiranju, dizajniranju i testiranju sigurnosti aplikacija [3, str. 57], [15]. Ovaj standard definira funkcionalne i nefunkcionalne sigurnosne kontrole uz fokus na web aplikacije. ASVS navodi sigurnosne kontrole koje pokrivaju arhitekturu, autentikaciju, kontrolu pristupa, validaciju podataka, kriptografiju, zaštitu podataka itd.

Kao dio sigurnosnih zahtjeva potrebno je navesti popis regulativa, standarada i politika koje je potrebno zadovoljiti (npr. HIPAA¹, PCI DSS², FFIEC³, GDPR⁴, itd.) [12, str. 22]. Ako postoji potreba za usklađenost s regulativama i standardima, potrebno je definirati sve zahtjeve koji proizlaze iz njih.

3.2. Utvrđivanje prihvatljivih granica kvalitete

S ciljem povećanja kvalitete softvera i smanjenja rizika, organizacije mogu definirati granice kvalitete (engl. *quality gates*). Politikom se određuju metrike koje se uzimaju u obzir, i za svaku metriku se određuje koji su rezultati prihvatljivi, a koji zahtijevaju obavezno ispravljanje prije nego se kod može pustiti u primjenu. Radi se o jednostavnom i učinkovitom načinu kontroliranja količine poznatih pogrešaka u softveru.

Primjeri metrika koje se mogu uzimati u obzir:

- broj upozorenja kompajlera
- broj kršenja pravila u statičkoj analizi koda
- broj kršenja prihvaćenog stila kodiranja
- ciklometrička složenost koda (engl. *cyclomatic complexity*)

¹ Health Insurance Portability and Accountability Act

² Payment Card Industry Data Security Standard

³ Federal Financial Institutions Examination Council

⁴ General Data Protection Regulation

- pokrivenost koda (engl. *code coverage*)
- broj upozorenja otkrivenih pomoću skenera ranjivosti.

Prag pogreški (engl. *bug bar*) je vrsta granice kvalitete koja pomaže u procjeni kritičnosti pogreški i potencijalnih sigurnosnih propusta [16, str. 30]. Tokom razvojnog procesa, potrebno je donositi odluke o tome da li se određene pogreške moraju ispraviti ili su prihvatljive. Može biti poželjno ne ispraviti pogrešku jer je uklanjanje presloženo ili bi trajalo predugo. Pragom pogreški se na samom početku procesa razvoja utvrđuje koje vrste pogreški se moraju obavezno ispraviti kako bi se izbjegla konfuzija tokom razvojnog procesa [16, str. 74].

Sve uočene pogreške se klasificiraju prema jačini (engl. *severity*), a prag pogreški određuje koju jačinu pogreška smije imati da bi bila prihvatljiva. Na primjer, pogreška se može klasificirati kao kritična, važna umjerena ili slaba, a samo pogreške ocijenjene kao slabe mogu biti prihvatljive, to jest ne mora ih se nužno ispraviti. Kriteriji za ocjenu jačine [17], [18]:

- Kritične pogreške: Ranjivosti čije iskorištavanje ne uzrokuje upozorenje. Ranjivosti zbog kojih udaljeni napadač lako može iskoristiti i dovesti do kompromisa sustava bez potrebe za interakcijom korisnika. Ranjivosti koje mogu iskoristiti crvi.
- Važne pogreške: Ranjivosti čije iskorištavanje uzrokuje upozorenje. Ranjivosti koji lako mogu ugroziti povjerljivost, integritet ili dostupnost resursa. Ranjivosti koje lokalnim korisnicima omogućavaju eskalaciju privilegija. Ranjivosti koje omogućuju neautenticiranim korisnicima dohvaćanje podataka koji bi trebali biti zaštićeni. Ranjivosti koje dopuštaju da autenticirani korisnici izvršavaju proizvoljni kod. Ranjivosti koje korisnicima omogućuju uskraćivanje usluge.
- Umjerene pogreške: Ranjivosti koje je teže iskoristiti, i samo pod određenim neočekivanim okolnostima. Ranjivosti koje za iskorištavanje zahtijevaju određene scenarije, konfiguracije sustava ili druge preduvjete. Ove ranjivosti mogu imati i kritične posljedice, ali postoji mala vjerojatnost za iskorištavanje.

- Slabe pogreške: Ostali propusti koja imaju utjecaj na sigurnost. Ranjivosti za koje se smatra da postoji vrlo malo vjerojatnosti da bi ih se moglo iskoristiti, ili kod kojih bi iskorištavanje imalo zanemarive posljedice. Ranjivosti čije iskorištavanje ima privremene posljedice, poput nekih vrsta uskraćivanja usluge.

Osim klasifikacije prema jačini, moguće je koristiti bodovanje prema CVSS⁵ sustavu. CVSS sustav boduje ranjivosti numeričkim rezultatom u rasponu od 0,0 (bez rizika) do 10,0 (najveći rizik). Sustav u obzir uzima [17]:

- Vektor napada: Lokalni, mrežni ili fizički.
- Složenost napada: Koliko je teško izvesti napad i koji su faktori potrebni za njegov uspjeh.
- Interakcija korisnika: Da li u napadu mora sudjelovati čovjek ili je napad moguće automatizirati.
- Potrebne privilegije: Nikakve, niske ili visoke.
- Opseg: Da li se napadom može utjecati na resurse s različitom razinom autoriteta.
- Povjerljivost: Mogu li se podaci otkriti neautoriziranim strankama, i koji je utjecaj.
- Integritet: Da li u napadu može doći do narušavanja integriteta podataka, i koji je utjecaj.
- Dostupnost: Utjecaj na dostupnost podataka ili usluge ovlaštenim korisnicima.

Označavanjem pogrešaka s ocjenom jačine i CVSS bodovima u sustavu za praćenje grešaka olakšava se upravljanje pogreškama i njihova prioritizacija, i tako smanjuje vjerojatnost da kritične i važne pogreške ostanu neispravljene [16, str. 74].

⁵ Common Vulnerability Scoring System: <https://nvd.nist.gov/vuln-metrics/cvss>

3.3. Procjena rizika

Procjena rizika se provodi s ciljem utvrđivanja razine ranjivosti sustava. Procjena obično uključuje upitnik koji ispunjavaju članovi tima [3, str. 140], [10, str. 163]. Upitnik se sastoji od pitanja kao što su:

- Korisnici aplikacije: Tko su korisnici aplikacije i na koji način joj pristupaju? Koji se mehanizmi autentifikacije i autorizacije koriste? Da li postoje anonimni korisnici? Da li postoje različite razine autorizacije?
- Podaci: Kakve vrste podataka se koriste u aplikaciji? Da li se koriste osobni podaci? Da li se i kako upravlja s osjetljivim i povjerljivim podacima?
- Arhitektura: Gdje će se izvršavati kod? Da li se kod ili dio koda izvršava na klijentskom računalu? Koji se programski jezik i razvojni okvir koriste? Da li su članovi organizacije upoznati s korištenim jezikom i razvojnim okvirom? Koje sigurnosne mogućnosti pruža razvojni okvir?
- Proces: Da li postoje automatizirani alati poput SAST alata i upravljanja ovisnostima za korišteni programski jezik i razvojni okvir?

Odgovori na upitnik služe kako bi se uočili rizici i slabosti rano u razvojnem ciklusu sustava. Ovisno o uočenim rizicima određuje se razina utjecaja aplikacije na sigurnost i privatnost, o čemu ovisi koje daljnje sigurnosne mjere moraju biti provedene (modeliranje prijetnji, pregled dizajna, sigurnosni pregled koda, penetracijsko testiranje, razina testiranja Fuzz metodom, itd.) [10, str. 79].

4. Analiza i dizajn

4.1. Primjena sigurnog dizajna

Sigurnost mnogih sustava narušena je zbog grešaka uzrokovanih nedostacima dizajna. Primjenom načela sigurnog dizajna, moguće je značajno smanjiti broj i utjecaj sigurnosnih proboja.

Ova načela definiraju učinkovite prakse primjenjive na razini arhitekture softvera, bez obzira na korištene tehnologije. Načela ne jamče sigurnost i ponekad mogu biti u suprotnosti, ali treba ih koristiti kao smjernice za poboljšavanje sigurnosti softvera. Većina ovih načela je poznata više desetljeća i još uvijek su relevantna [19].

Obrana u dubinu (*Slojevita obrana*, engl. *defense in depth*): Sustav treba imati neovisne sigurnosne slojeve tako da napadač mora zaobići više sigurnosnih mjera da bi napad uspio. Pojedinačne točke potpunog kompromisa uklanjaju se ili ublažavaju ugradnjom niza ili više slojeva sigurnosnih zaštitnih mjera i mjera suzbijanja rizika. Različite obrambene strategije dovesti će do toga da ako se jedan sloj obrane pokaže neadekvatnim, drugi sloj obrane može spriječiti potpuni prodor [20], [21].

Ekonomičnost mehanizma (engl. *economy of mechanism*): Dizajn bi trebao biti što jednostavniji. S povećanom složenošću softverskog dizajna i koda, povećava se vjerojatnost većeg broja ranjivosti. Otkrivanje pogreški dizajna i implementacije koje rezultiraju neželjenim pristupnim stazama učinkovitije je što je dizajn jednostavniji. Jednostavnijim dizajnom osigurava se manja površina napada [20], [22]. Načelo je također poznato kao KISS.

Zadana sigurnost (engl. *fail-safe defaults*): Ako subjektu nije izričito odobren pristup objektu, trebao bi mu biti uskraćen pristup tom objektu. Odluke o pristupu treba temeljiti na dopuštenju, a ne na uskraćivanju pristupa. Dizajn se mora temeljiti na argumentima zašto bi objekti trebali biti dostupni, a ne zašto ne bi trebali. Pogreška u implementacije mehanizma koji daje izričito dopuštenje će rezultirati pogrešnim odbijanjem pristupa, što je sigurna situacija koju je lako otkriti. U suprotnom, došlo bi do potencijalno neopaženog i neovlaštenog pristupa objektu [20], [22].

Potpuno posredovanje (engl. *complete mediation*): Mora se provjeriti ovlaštenje pristupa svakom objektu prilikom svakog zahtjeva. Ovo načelo podrazumijeva

postojanje metode identifikacije izvora svakog zahtjeva. Rezultati provjere pristupa često se pamte, npr. u predmemoriji. U tom je slučaju potrebno primijeniti mehanizam koji će u slučaju promjene dozvola očistiti zapamćeni rezultat. Ovo načelo osigurava da se autorizacija ne zaobiđe u sljedećim zahtjevima objekta od strane subjekta [20], [22].

Otvoreni dizajn (engl. *open design*): Dizajn ne bi trebao biti tajna. Sigurnost mehanizma ne bi trebala ovisiti o tajnosti njegovog dizajna. Detalji implementacije dizajna trebaju biti neovisni o samom dizajnu koji bi trebao ostati otvoren. Ako sigurnost ovisi o tajnosti dizajna, a detalji dizajna procure, cijeli sustav je ugrožen. Ako su tajne odvojene od mehanizma, npr. s tajnim ključem, tada curenje ključa utječe na samo jednog korisnika. Primjena ovog načela omogućava pregled dizajna bez brige o ugrožavanju sigurnosti [20], [22].

Razdvajanje dužnosti (engl. *separation of duties*, *Separation of privilege*, *Compartmentalization principle*): Kritični zadaci trebaju se rastaviti u zasebne elemente koje obavljaju zasebni subjekti. Uspješan završetak pojedinačnog zadatka treba ovisiti o dva ili više uvjeta koja je potrebno ispuniti, a samo jedan od uvjeta neće biti dovoljan da sam zadatak izvrši [20], [21].

Najmanja privilegija (engl. *least privilege*): Svaki subjekt u sustavu treba imati najmanju razinu prava pristupa koja je potrebna za izvršavanje zadatka. Pravo pristupa se subjektu mora dati samo za minimalno vrijeme koje je potrebno za izvršavanje zadatka. Ovaj princip ograničava štetu koja može rezultirati u slučaju pogreške ili iskorištene ranjivosti [20], [22].

Najmanji zajednički mehanizam (engl. *least common mechanism*): Dijeljenje mehanizama između subjekata u sustavu treba biti svedeno na minimum. Svaki zajednički mehanizam povećava rizik jer predstavlja potencijalni put informacija između subjekata ili sredstvo ometanja. Na primjer, skriveni kanali se mogu pojaviti kada se zasebni procesi izvršavaju na dijeljenim računalnim resursima. U ovom slučaju načelo se primjenjuje izolacijom procesa korištenjem virtualizacije ili zaštićene okoline (engl. *sandbox*, *jail*) [20], [22].

Psihološka prihvatljivost (engl. *psychological acceptability*): Sigurnosna funkcionalnost treba biti jednostavna za upotrebu i transparentna prema korisniku.

Sigurnosni mehanizmi ne bi trebali korisniku otežati pristup resursu. Ako korisnici razumiju ciljeve sigurnosnih mehanizama, pogreške će se svesti na najmanju moguću mjeru. Cilj ovog načela je povećati usvajanje i upotrebu sigurnosnih funkcionalnosti u softveru [20], [22].

Faktor rada (engl. *work factor*): Jače sigurnosne mjere otežavaju rad napadača. Na primjer, dulje lozinke i veći ključevi za enkripciju povećavaju faktor rada napadača povećanjem broja pokušaja potrebnih da ih se pogodi [21], [22].

Bilježenje kompromitiranja (engl. *compromise recording*): Svaki pristup ili pokušaj pristupa objektu treba biti zabilježen. Bilježenje pristupa može pomoći otkriti neautorizirani pristup [22].

Najslabija veza (engl. *weakest link*): Sustav je siguran koliko i njegov najslabiji sigurnosni mehanizam. Otpornost softvera na napade uvelike će ovisiti o zaštiti njegovih najslabijih komponenti, bilo da je to kod, infrastruktura ili ljudski faktor. Dobro je pobrinuti se prije za najozbiljnije rizike, umjesto za rizike koje je lakše najlakše umanjiti [20].

Korištenje postojećih komponenti (engl. *leveraging existing components*): Kada je moguće, potrebno je koristiti postojeće isprobane i testirane komponente (npr. kod, biblioteke, funkcionalnosti) da bi se izbjeglo uvođenje novih površina napada ili novih ranjivosti [20].

4.2. Pregled arhitekture i dizajna

Dizajn i arhitekturu aplikacije treba dokumentirati u obliku dijagrama, tekstualnih i drugih dokumenata. Dokumentaciju treba pregledati i provjeriti da dizajn i arhitektura zadovoljavaju odgovarajuću razinu sigurnosti i kriterije definirane u zahtjevima [12, str. 35].

Provedba ove aktivnosti omogućuje pronalazak i ispravljanje propusta u arhitekturi i dizajnu prije nego započne faza implementacije, te prepoznavanje potencijalnih ranjivosti prije nego što se mogu iskoristiti i prije nego što ispravak postane znatno zahtjevniji zahvat. Pregledom arhitekture se također mogu identificirati potencijalne promjene čijom se primjenom može smanjiti površina napada sustava, čime se povećava sigurnost korisnika i sustava [10, str. 84].

Prilikom pregleda arhitekture i dizajna potrebno je uzeti u obzir sljedeća područja [10, str. 84]:

- Kritični dijelovi aplikacije: Pregled provjere autentičnosti, autorizacije, provjere unosa, kriptografija, upravljanje iznimkama, logiranje, itd.
- Analiza sloj-po-sloj: Pregled logičkih slojeva aplikacije (npr. slojevi prezentacije, poslovanja i pristupa podacima) i procjena sigurnosnih odluka unutar njih.
- Okruženje i infrastruktura: Pregled dizajna u odnosu na ciljno okruženje i infrastrukturu. Potrebno je uzeti u obzir sigurnosni utjecaj infrastrukturnog sloja.
- Usklađenost s regulatornim zahtjevima i politikama.

Ovisno o projektu i organizaciji, pregled mogu obavljati sigurnosni inženjeri, arhitekti i programeri, a moguće je da ga obavi i vanjski savjetnik.

Rezultat pregleda je dokument koji sadrži nedostatke u ispunjavanju sigurnosni zahtjeva koji nisu zadovoljeni dizajnom te druge uočene nedostatke i plan njihovog ublažavanja. Na osnovi rezultata pregleda, arhitekti sustava mogu ispraviti dizajn.

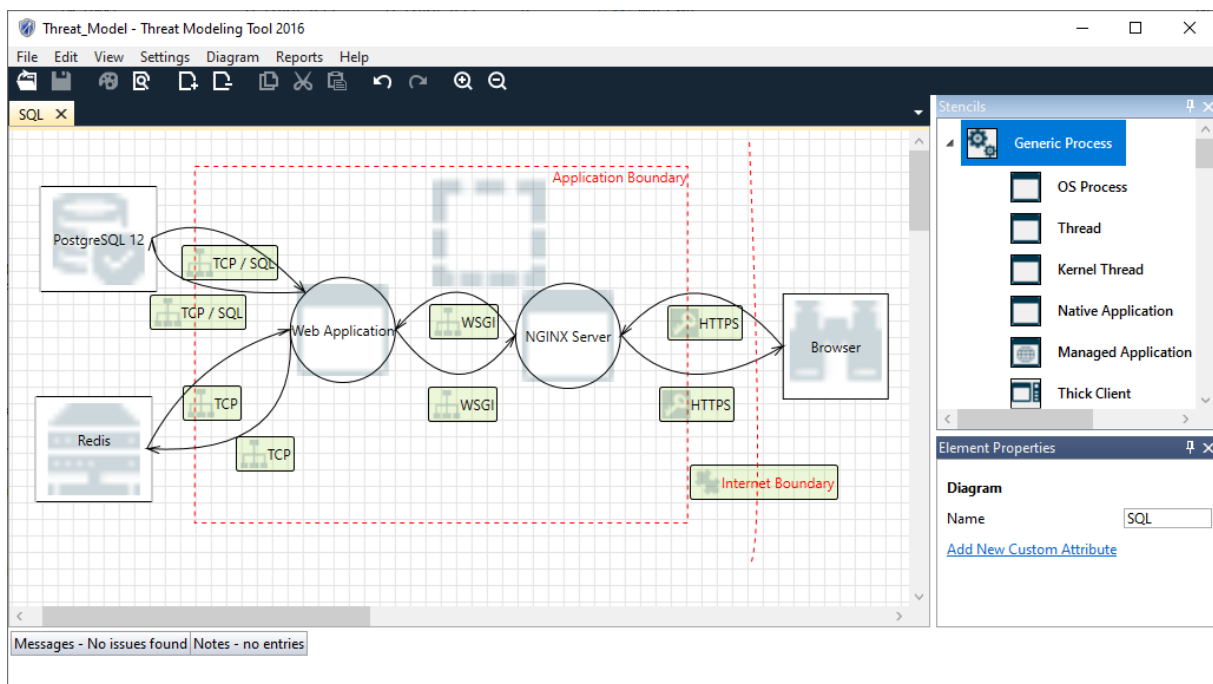
4.3. Modeliranje prijetnji

Modeliranje prijetnji je tehnika identifikacije, procjene i upravljanja prijetnjama sustava, manama dizajna i mjerama ublažavanja rizika. U modeliranje prijetnji mogu biti uključeni vlasnici proizvoda, arhitekti, programeri, sigurnosni i testni inženjeri [13, str. 25]. Ovom aktivnošću se povećava svijest o sigurnosti i povezuje tehnički rizik s utjecajem na poslovanje [16, str. 101].

Modeliranje prijetnji započinje crtanjem dijagrama toka podataka. Razumijevanje dizajna aplikacija ključni je preduvjet za modeliranje prijetnji aplikacije. To je nužno kako bi bilo moguće nacrtati točan dijagram toka podataka i identificirati sve rizike.

Dijagram toka podataka predstavlja sliku sustava ili dijela sustava na kojemu se provodi modeliranje, s istaknutim elementima na kojima se podaci unose, mijenjaju, prenose ili pohranjuju, i vezama između tih elemenata.

Dijagram toka podataka također treba sadržavati granice povjerenja. Granice povjerenja su mjesta u sustavu gdje se podaci kreću između zona različitih privilegija ili povjerenja. Ovisno o razini apstrakcije, mogu se uspostaviti granice između organizacija, podatkovnih centara ili mreža, procesa, slojeva aplikacije i slično. Uobičajena konvencija je crtanje granica na dijagramu kao isprekidane crte koje označavaju različite zone povjerenja [3, str. 135].



Slika 2: Dijagram toka podataka u alatu Microsoft Threat Modeling Tool

Nakon što je dijagram gotov, slijedi identifikacija prijetnji. Prijetnje se otkrivaju detaljnim pregledom popisa svih elemenata dijagrama toka podataka. Svaki element treba razmotriti kao potencijalni predmet napada [16, str. 119]. Kako bi se razmotrile različite vrste sigurnosnih prijetnji i rizika preporučuje se strukturirani pristup korištenjem spiska. Jedan od često korištenih pristupa je STRIDE. STRIDE označava [3, str. 139]:

- *Spoofing* (lažiranje): prijetnja zbog koje se napadač može predstaviti kao nešto ili netko drugi (npr. drugi korisnik, druga domena ili izvršna datoteka).
- *Tampering* (neovlašteno mijenjanje): prijetnja koja uključuje zlonamjernu neovlaštenu izmjenu podataka ili koda (npr. izmjena izvršne datoteke ili podataka u mrežnoj komunikaciji).

- *Repudiation* (osporavanje): prijetnja zbog koje napadač može poreći da je izvršio radnju koju druge stranke ne mogu dokazati suprotno (npr. nemogućnost dokazivanja pristupanja podacima).
- *Information Disclosure* (otkrivanje informacija): prijetnja koja uključuje otkrivanje informacija subjektima koji ne bi trebali imati pristup do njih (npr. neautorizirani pristup datoteci, napad „čovjek u sredini“).
- *Denial of Service* (uskraćivanje usluga): prijetnja koja uključuje uskraćivanje ili degradaciju usluge važećim korisnicima (npr. rušenje web poslužitelja, blokiranje mrežnog prometa).
- *Elevation of Privilege* (povisivanje privilegija): prijetnje uključuje dobivanje privilegija bez odgovarajuće autorizacije (npr. dobivanje administratorskih privilegija iskorištavanjem propusta u kodu).

Identificirane prijetnje se popisuju u tablicu, uz oznaku vrste prijetnje prema STRIDE klasifikaciji. Za svaku prijetnju se određuje veličina potencijalne štete, vjerojatnost događanja i rizik prema formuli [16, str. 121]:

$$\text{Rizik} = \text{Potencijalna šteta} \times \text{Vjerojatnost događaja}$$

Kako bi se smanjili ili uklonili rizici, potrebno je primijeniti ublažavanje rizika. Na svaku od identificiranih prijetnji primjenjuje se jedna od sljedećih strategija:

- primjena sigurnosne mjere
- uklanjanje elementa iz sustava
- redizajn sustava
- upozoravanje korisnika (vlasnika)
- prihvaćanje rizika.

Sigurnosne mjere koje se obično koriste ovisno o STRIDE klasifikaciji su [16, str. 126]:

- Lažiranje: autentikacija, PKI, digitalni certifikati, sažetci podataka.

- Neovlašteno mijenjanje: kontrola pristupa, digitalno potpisivanje, sažetci podataka.
- Osporavanje: autentikacija, logiranje i revizija, digitalni certifikati i potpisi.
- Otkrivanje informacija: Kontrola pristupa, enkripcija.
- Uskraćivanje usluga: autentikacija, ograničavanje brzine, filtriranje i granična zaštita.
- Povećavanje privilegija: autentikacija, autorizacija, princip najmanje privilegije.

Modele prijetnji bi trebalo razviti što je prije moguće u razvojnem procesu, i treba ih ponovno razmotriti kako se aplikacija razvija [12, str. 16].

Modele prijetnji je moguće koristiti pri pregledu koda i testiranju aplikacije kako bi se pregledali i testirali svi dijelovi kritični za sigurnost sustava [16, str. 128].

5. Implementacija

5.1. Primjena praksi sigurnog kodiranja

Na osnovi najboljih praksi pisanja sigurnog koda, organizacija treba propisati smjernice za sigurno kodiranje (engl. *secure coding*). Provođenje smjernica treba kontrolirati kroz pregled koda i pomoću alata za analizu koda. Smjernice mogu sadržavati spisak uvjeta koje svaki kod mora zadovoljiti. OWASP vodič o praksi sigurnog kodiranja (engl. *OWASP Secure Coding Practices Quick Reference Guide*) definira uvjete iz sljedećih područja [4, str. 5]:

- Validacija ulaznih podataka: Svi ulazni podaci trebaju biti validirani. Provjeravaju se kodiranje, tip podatka, vrijednost ili opseg vrijednosti, format, itd. U slučaju neispravnih podataka, podaci se odbijaju.
- Kodiranje izlaznih podataka: Korištenje standarda za kodiranje, saniranje nepouzdanih podataka i sigurno kodiranje znakova (npr. *HTML entity encoding*).
- Autentifikacija i upravljanje lozinkama: Autentifikacija, spremanje, promjena lozinke i resetiranje lozinke se mora izvršavati na siguran i testiran način. Moraju se koristiti odgovarajući kriptografski i *hash* algoritmi.
- Upravljanje sesijama: Da li se identifikator sesije generira i šalje na siguran način?
- Kontrola pristupa: Onemogućavanje pristupa podacima i izvršavanju radnji neautoriziranim korisnicima na siguran način. Onemogućavanje pristupa resursima izvan kontrole aplikacije.
- Kriptografske prakse: Moraju se koristiti sigurni kriptografski algoritmi. Kriptografski tajni ključevi moraju biti zaštićeni. Način generiranja slučajnih vrijednosti mora biti siguran.
- Upravljanje pogreškama i logiranje: Obavijesti o pogreškama ne smiju korisniku otkrivati osjetljive podatke. Poruke o greški moraju sadržavati samo generičke poruke. Potrebno je logirati sve greške u autorizaciji, autentifikaciji, validaciji podataka, sistemske pogreške itd. Logovi moraju

sadržavati sve potrebne kritične podatke, i moraju biti pohranjeni na siguran način.

- Zaštita podataka: Uključuje korištenje načela najmanjih privilegija, enkripcija osjetljivih podataka, onemogućavanje spremanja osjetljivih podataka u predmemoriju (engl. *cache*), zaštitu privremenih podataka, itd.
- Sigurnost komunikacije: Svi osjetljivi podaci se moraju prenositi isključivo kriptiranim komunikacijskim kanalima.
- Konfiguracija sustava: Moraju se koristiti najnovije odobrene verzije servisa, radnih okvira i komponenti. Servisi i procesi moraju imati ograničena prava pristupa resursima. Razvojno okruženje mora biti izolirano od produkcijskog.
- Sigurnost baze podataka: Upiti moraju biti parametrizirani. Pristup bazi se mora autentificirati. Aplikacija mora imati najmanje moguće privilegije nad bazom.
- Upravljanje datotekama: Zaštita datoteka od neovlaštenih promjena i izvršavanja.
- Upravljanje memorijom: Validacija ulaznih i izlaznih podataka, provjera veličine spremnika, izbjegavanje korištenja ranjivih funkcija, zatvaranje resursa (datoteka, mrežnih veza itd.).
- Opće prakse kodiranja: Korištenje sistemskih API-ja umjesto izvršavanja naredbi. Sinkronizacija međusobnim isključivanjem radi izbjegavanja stanje trke (engl. *race condition*). Eksplicitna inicijalizacija vrijednosti varijabli. Podizanje privilegija samo u trenutku kada je to nužno. Izbjegavanje aritmetičkih pogrešaka i prelijeva cijelih brojeva.

Osim OWASP vodiča o praksi sigurnog kodiranja, postoje i druge slobodno dostupne smjernice za sigurno kodiranje [3, str. 162]:

- *Secure coding guidelines for .NET⁶ (Microsoft)*

⁶ <https://docs.microsoft.com/en-us/dotnet/standard/security/secure-coding-guidelines>

- *WebAppSec / Secure Coding Guidelines*⁷ (Mozilla)
- *Secure Coding Guidelines for Java SE*⁸ (Oracle)
- *SEI CERT Coding Standards*⁹.

Kao rezultat praćenja smjernica za sigurno kodiranje, kod će biti „čišći“, razumljiviji, pregledniji i sigurniji [3, str. 162].

Kod koji je pretjerano složen teško je procijeniti ručnim pregledom koda, a i statička analiza je manje učinkovita [23], stoga će u kodu koji je „čist“, jednostavan i dobro održavan biti lakše uočiti propuste.

Razvojni inženjeri bi trebali redovito prisustvovati treningu o tehnikama sigurnog kodiranja, uključujući kako izbjeći uobičajene ranjivosti.

5.2. Pisanje testova

U modernom razvojnom procesu softvera, zbog novih zahtjeva, promjena zahtjeva, prilikom refaktoriranja i odgovora na povratne informacije, na kodu se stalno rade iterativne izmjene. Izmjene mogu uzrokovati slučajne promjene željenog ponašanja sustava, to jest stvoriti defekte, uključujući i sigurnosne ranjivosti. Kako bi se to izbjeglo, programeri razvijaju automatizirane regresijske testove [3, str. 193].

Programeri najčešće pišu pozitivne testove (još se nazivaju i *happy path* testovima). Ovi testovi testiraju da sustav ima očekivano ponašanje s ispravnim ulaznim podacima, to jest da kod radi ono što bi trebao raditi. Ovi testovi nisu dovoljni da bi pronašli regresijske pogreške i sigurnosne propuste [23], [24]. Takvi problemi se najčešće otkrivaju pomoću dobro napisanih negativnih testova. Negativni testovi testiraju da sustav ima očekivano ponašanje s neispravnim ulaznim podacima, to jest da kod ne radi ono što ne smije raditi.

⁷ https://wiki.mozilla.org/WebAppSec/Secure_Coding_Guidelines

⁸ <https://www.oracle.com/technetwork/java/seccodeguide-139067.html>

⁹ <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>

Negativni testovi trebaju izazvati očekivane pogreške, to jest neuspjeh ili odbacivanje „korisničkog zahtjeva“. Negativni testovi trebaju testirati svaku vrstu neispravnog ulaznog podatka [23].

Pokrivenost koda (engl. *code coverage*) je naziv metrike koja pokazuje koja je količina koda pokrivena testovima, to jest koliko koda se izvršava tokom testiranja. Alati za identifikaciju pokrivenosti koda obično daju rezultate o pokrivenosti modula, klase, datoteka i pojedinih redaka datoteke. Rezultati koje ovi alati daju je pouzdanije mjerilo pokrivenosti od samog broja testnih slučajeva. Pokrivenost koda također otkriva dijelove koda koji nisu dovoljno testirani.

Pokrivenost koda ne govori ništa o samoj kvaliteti testova, ali ipak može imati pozitivan utjecaj na razvoj softvera. Kontrolom pokrivenosti koda se može osigurati obavezno pisanje testova. Kvaliteta testova se provjerava u fazi pregleda koda.

Izrada testova povećava ukupno početno vrijeme razvoja, ali se njima postiže veća kvaliteta proizvoda i na kraju će konačni rezultat biti bolji [25]. Svi testovi moraju završiti uspješno da bi kod bio prihvaćen i završio fazu implementacije.

5.3. Pregled koda

Uobičajena je praksa da se programeri oslanjaju na pregled koda (engl. *code review*) drugih programera kako bi identificirali nedostatke i poboljšali kvalitetu koda. Ciljevi pregleda koda su:

- pronalazak propusta
- provođenje prihvaćenih konvencija, stilova i najboljih praksi kodiranja
- prijenos tehničkog znanja i poznavanje detalja projekta između članova tima
- stvaranje zajedničkog osjećaja vlasništva između članova tima
- poticanje odgovornosti.

Pregled koda može identificirati propuste u ispunjavanju zahtjeva, propuste u dizajnu i arhitekturi, propuste u dokumentaciji te propuste u samom kodu.

Pregled koda može biti nužan za zadovoljavanje propisa o usklađenosti (npr. PCI DSS).

Negativne strane pregleda koda:

- trošak zbog vremena koje programer mora potrošiti na pregled koda
- neke neispravnosti je teško otkriti vizualnom inspekcijom (npr. curenje memorije)
- mogući konflikti između članova tima.

Sigurnosni pregled koda proširenje je standardne prakse pregleda koda gdje se u postupku pregleda uključuju sigurnosna razmatranja, poput sigurnosnih standarda organizacije, na čelo donošenja odluka. Kako bi recenzenti učinkovito ocijenili kod, nužno je da imaju potrebne vještine i znanje sigurnog kodiranja, kao i ispravan sigurnosni kontekst prilikom pregleda koda.

Razina pregleda koda ovisiti će ovisno o poslovnim ili regulatornim potrebama softvera, veličini organizacije koja razvija softvera i vještinama osoblja. [26, str. 12]

Kako bi se povećala učinkovitost pregleda koda i osiguralo da se postignu ciljevi, organizacije mogu uvesti smjernice za pregled koda kojima će se utvrditi:

- ciljevi pregleda koda
- tko provodi pregled koda, i koje vještine i znanja te osobe moraju posjedovati
- kada se obavlja pregled koda
- zahtjevi koje svaki kod mora zadovoljiti
- aspekte na koje treba obratiti pažnju prilikom pregleda koda
- način komunikacije između autora i recenzenta koda.

Neke propuste, poput kršenja prihvaćenih konvencija kodiranja, je moguće vrlo učinkovito identificirati automatiziranim alatima (SAST, linteri). Stoga je bitno primijeniti automatizirane metode identifikacije propusta prije pregleda koda, tako da recenzent ne mora gubiti vrijeme na estetske probleme, nego se može fokusirati na funkcionalne propuste, i tako povećati učinkovitost procesa.

Povremeno se može izvršiti šetnja kroz kod (engl. *code walk-through*), metoda pregleda koda na visokoj razini pri kojoj se objašnjava logika i tok koda. Za potrebe

procjene sigurnosti ovaj pregled koda provode sigurnosni inženjeri uz sudjelovanje razvojnih inženjera [12, str. 36].

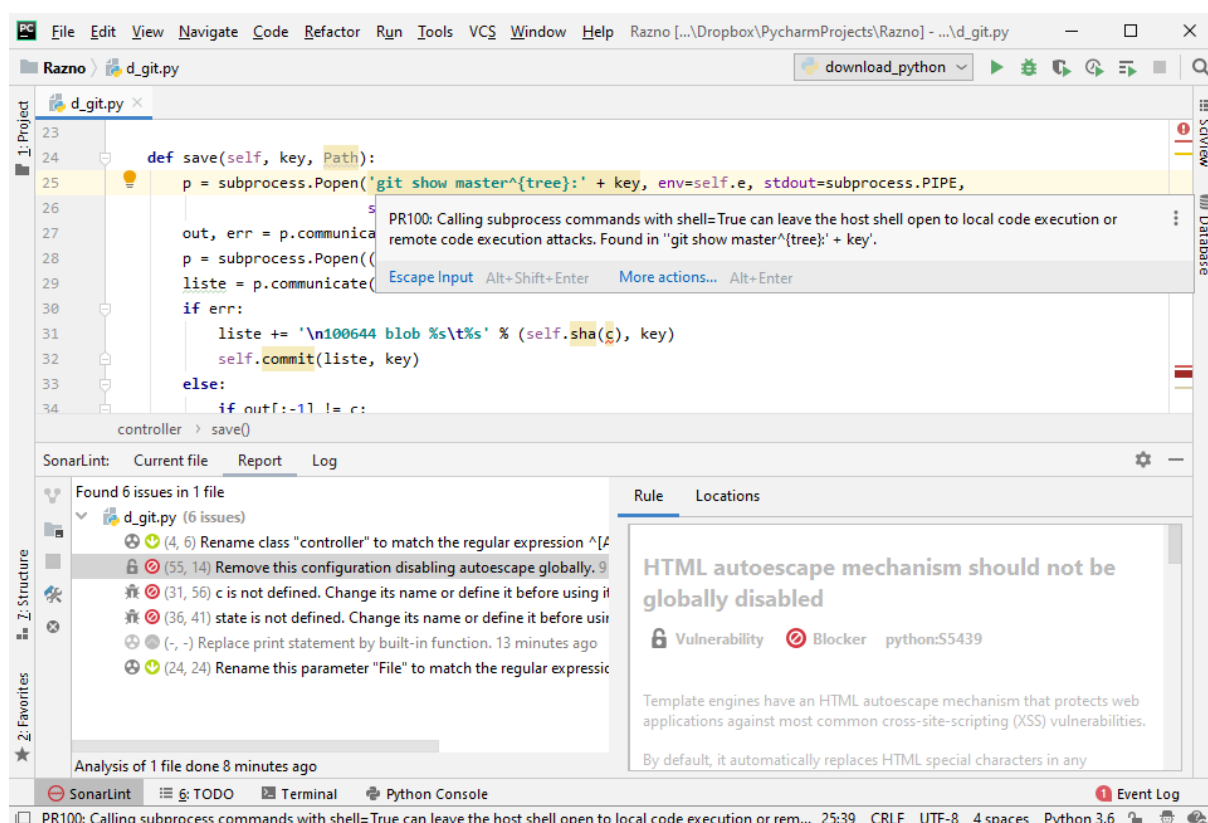
5.4. Statička analiza koda

Statička analiza koda je automatizirana metoda pregleda koda. Alati za statičku analizu koda mogu otkriti pogreške u kodu, sigurnosne propuste, neispravan stil kodiranja, duplicirani kod, mrtvi kod, ranjivosti u ovisnim komponentama, te mogu utvrditi kompleksnost koda i druge metrike. Analiza se obavlja statički, to jest bez izvršavanja samog koda. Alati koriste metode poput analize toka izvršavanja, toka podataka, heuristike i analize uzoraka [3, str. 171].

Alati se mogu podijeliti po:

- Svrhovitosti: generički ili specijalizirani za određenu svrhu (npr. sigurnost ili stil kodiranja).
- Programskom jeziku: višejezični ili specijalizirani za određeni programski jezik.
- Način korištenja: integracijom u IDE (integrirano razvojno okruženje), dio CI/CD procesa, pomoću naredbenog retka, itd.

Korištenje alata za analizu koda unutar IDE-a omogućuje programeru da uoči određene pogreške za vrijeme pisanja, pohranjivanja ili kompajliranja koda (Slika 3). Ovi alati mogu biti već ugrađeni u IDE ili se mogu instalirati kao dodatci. Ovi alati često ne mogu vršiti dubinsku analizu koda, ali mogu smanjiti broj čestih propusta na razini modula [3, str. 171]. Neki alati, osim što upozoravaju programera na propust, imaju i mogućnost da automatski isprave propust.



Slika 3: Upozorenja alata za statičku analizu koda unutar IDE sučelja

Statička analiza koda je učinkovit način provođenje politike kodiranja. Automatizirani alat najbolji je način za provođenje politika poput zabrane određenih API-ja ili načina korištenja određenih API-ja [16, str. 148]. Provođenje politike o stilu kodiranja pomoću automatiziranog alata olakšava posao programeru i recenzentu koda tako što im omogućavaju da se fokusiraju na važnije stvari.

Statičko testiranje sigurnosti aplikacije (SAST, engl. *static application security testing*) je statička analiza koda kojoj je glavni cilj otkrivanje ranjivosti. SAST alatima se provodi testiranje bijele kutije (engl. *white box*), jer je potreban izvorni kod. Probleme koje ovi alati otkrivaju, najčešće je lako ukloniti.

Neki od problema koje SAST alati mogu otkriti su:

- Nedostatak validacije podataka i različite vrste umetanja koda.
- Korištenje nesigurnih funkcija i API-ja.
- Korištenje slabih kriptografskih funkcija i algoritama sažimanja.

- Potencijalne ranjivosti uskraćivanja usluge.

Za razliku od ljudskih recenzenata, alati su konzistentni i mogu redovito analizirati velike količine koda. Iako ne mogu pronaći sve, vrlo će pouzdano pronaći neke potencijalno kritične propuste [3, str. 169]. Ipak, statička analiza koda ne može zamijeniti ručni pregled koda, nego je komplementarna sigurnosna mjera kojom se povećava učinkovitost cijelog razvojnog procesa.

6. Verifikacija i validacija

6.1. Sigurnosno testiranje

Nakon što programeri testiraju napisani kod pomoću *unit* testova u fazi implementacije, potrebno je provesti testiranje na aplikaciji kao cjelini. Integracijsko testiranje je faza testiranja u kojoj se provjerava da skup sučelja i modula ima očekivano ponašanje unutar aplikacije. Sistemsko testiranje je faza testiranja u kojoj se provjerava da aplikacija ispunjavanja zahtjeve. Sigurnosnim testovima u ovim fazama testiranja se provjeravaju sigurnosne funkcionalnosti aplikacije u cjelini, te izloženost ranjivostima na razini aplikacije [12, str. 27].

Sigurnosnim testiranjem potrebno je testirati sve potencijalno ranjive dijelove sustava.

Sigurnosno testiranje treba [27, str. 14]:

- utvrditi da su ispunjeni funkcionalni i nefunkcionalni sigurnosni zahtjevi
- utvrditi da aplikacija ne sadrži uobičajene ranjivosti (poput OWASP top 10 i CWE top 25 ranjivosti)
- utvrditi da su uklonjeni nedostaci identificirani u ranijim aktivnostima razvojnog procesa (modeliranje prijetnji, statička analiza koda i pregled koda)
- utvrditi da ključni dijelovi sustava identificirani u modeliranju prijetnji funkcioniraju kako je očekivano.

Sigurnosno testiranje obično izvode testni inženjeri u sklopu integracijskog i sistemskog testiranja. Testni inženjeri trebaju imati znanje o najčešćim ranjivostima, sigurnosnim kontrolama i mjerama za ublažavanje prijetnji, te tehnikama testiranja crne kutije i bijele kutije [12, str. 27].

Preduvjet za sigurnosno testiranje je dokumentacija sigurnosnih testnih slučajeva. Sigurnosni testni slučajevi se identificiraju i definiraju iz sigurnosnih zahtjeva te prethodno identificiranih i ispravljenih nedostataka (regresijski testovi). Testne slučajeve je moguće definirati u fazi prikupljanja zahtjeva, dizajna ili implementacije [12, str. 22], [13, str. 48].

Uz ručno testiranje, dobro je razvijati i koristiti automatizirane testove. Kada god je to moguće, za testni slučaj se implementira automatizirani test. Tako se omogućuje redovito i učinkovito izvršavanje regresijskog sigurnosnog testiranja.

6.2. Skeniranje ranjivosti

Skeniranje ranjivosti je automatizirana metoda otkrivanja ranjivosti sustava za vrijeme izvršavanja.

Skeniranjem ranjivosti na temelju potpisa (engl. *signature based*) identificiraju se *host* računala i njihove osobine (npr. operacijski sustav, otvoreni portovi, pokrenuti servisi), te pokušavaju identificirati njihove ranjivosti. Skeniranjem ranjivosti mogu se identificirati zastarjele verzije softvera, nedostajuće zakrpe, korištenje slabe enkripcije, pogrešne konfiguracije te pronaći odstupanja od sigurnosnih politika organizacije. Nakon što su identificirani podaci o operacijskom sustavu i softverskim aplikacijama, skener ih uspoređuje s podacima o poznatim ranjivostima pohranjenim u skenerovoj bazi podataka. [28, str. 27].

Osim skeniranja na temelju potpisa, neki skeneri ranjivosti pokušavaju simulirati stvarni napad koristeći poznate tehnike iskorištavanja ranjivosti, te u slučaju uspjeha prijavljuju pronađene ranjivosti [28, str. 28]. Korištenje ovih alata se još naziva dinamičko testiranje sigurnosti aplikacije (DAST, engl. *dynamic application security testing*).

DAST alatima moguće je otkriti propuste poput različitih vrsta umetanja koda (SQL, XML, LDAP, HTTP zaglavlja, itd.), *cross-site* skriptiranje, izloženost osjetljivih podataka, *clickjacking*, loša autentifikacija i kontrola pristupa, itd.

Interaktivno testiranje sigurnosti aplikacije (IAST, engl. *interactive application security testing*) je hibridni pristup testiranju koji kombinira SAST i DAST tehniku. Aplikacija se testira dinamički, uz stalno praćenje internog stanja aplikacije [3, str. 173], [13, str. 51].

6.3. Penetracijsko testiranje

Kako bi se detaljno ispitalo postojanje ranjivosti u kompleksnim aplikacijama, nije dovoljno koristiti automatizirane alate. Razumijevanje funkcionalnosti aplikacije omogućuje pronalazak ranjivosti koje generički alati ne mogu otkriti.

Penetracijsko testiranje je metoda ispitivanja u kojoj se sigurnost računalnog programa ili mreže ispituje simulacijom napada [10, str. 131]. U simulaciji napada često se koriste isti alati koje koriste stvarni napadači. Proces uključuje ispitivanje ranjivosti i otkrivanje metode kojom napadač može dobiti neovlašteni pristup sustavu ili podacima.

Da bi penetracijsko testiranje bilo uspješno i kvalitetno odrađeno, osoba koja provodi testiranje (pen-tester) mora imati visoku razinu stručnosti i iskustva. Ako unutar organizacije ne postoje osobe s odgovarajućim vještinama, unajmljuje se treća tvrtka kako bi obavila testiranje [16, str. 165].

Ovisno o informacijama o sustavu dostupnim pen-testeru, penetracijsko testiranje se dijeli na [1], [29]:

- Testiranje crne kutije (engl. *black box testing*): Testiranje se vrši bez poznavanja testiranog sustava.
- Testiranje bijele kutije (engl. *white box testing*): Pen-tester na raspolaganju ima detaljne informacije o testiranom sustavu kao što su izvorni kod aplikacija, IP adrese, URL-ovi, itd.
- Testiranje sive kutije (engl. *gray box testing*): Pen-tester ima djelomično znanje o testiranom sustavu, npr. može poznavati format unosa korisničkih podataka i očekivane metode validacije podataka i kako se obrađuju podaci [1], [12].

U kontekstu procesa razvoja softvera, obično se provodi testiranje bijele kutije ili sive kutije. Ovi testovi su učinkovitiji i daju preciznije i sveobuhvatnije rezultate od testiranja crne kutije [30, str. 3].

Za svaki penetracijski test određuje se opseg testiranja, to jest koje dijelove sustava je potrebno testirati. Opsegom se testiranje može ograničiti na vanjske perimetre (javno dostupne površine napada) ili unutarnje perimetre (površine napada unutar LAN mreže) [30, str. 4]. Opseg također uključuje koliko dugo traje testiranje: može se raditi o danima ili tjednima.

Penetracijsko testiranje se najčešće obavlja na kraju razvojnog procesa, prije nego se aplikacija stavlja u primjenu [3, str. 230].

6.4. Testiranje Fuzz metodom

Fuzz testiranje je učinkovita metoda sigurnosnog testiranja u kojoj se koriste neispravni ili nasumični ulazni podaci s ciljem izazivanja pogreške. Ovakvi testovi su korisni u pronalaženju sigurnosnih propusta jer se njima može testirati ogroman broj neočekivanih ulaznih podataka. Te podatke generiraju *fuzzing* alati (ili *fuzzeri*), alati koji su posebno dizajnirani za ovu metodu testiranja. Dio aplikacije koji je potrebno testirati fuzz metodom se određuje korištenjem modela prijetnji ili se testiraju najrizičnije komponente aplikacije. [10, str. 70]

Fuzzeri mogu biti [23]:

- Potpuno nasumični *fuzzeri*: Najjednostavniji *fuzzeri* koji generiraju stvarno slučajne podatke.
- Mutacijski ili glupi *fuzzeri*: Započinju s zadanim početnim uzorcima koje modificiraju određenim transformacijama. Omogućuju detaljnije testiranje od potpuno nasumičnih *fuzzera*.
- Generacijski ili pametni *fuzzeri*: Generiraju podatke prema detaljno zadanim pravilima. Omogućuju najdetaljnije testiranje ali zahtijevaju veći uloženi trud.

Ovom metodom moguće je otkriti *Zero Day* ranjivosti, to jest ranjivosti koje nisu poznate, pa ju koriste istraživači sigurnosti, a neki automatizirani alati za skeniranje ranjivosti pri skeniranju koriste *fuzzere* [3, str. 216]. Moguće je otkriti ranjivosti poput prelijeva spremnika (engl. *buffer overflow*), prelijeva cijelih brojeva (engl. *integer overflow*), korupcije memorije, umetanja SQL naredbi, *Cross Site Scripting*, udaljeno izvršavanje naredbi itd. [31]

6.5. Osiguravanje lanca dobavljača softvera

Korištenje biblioteka koda i radnih okvira otvorenog koda je danas vrlo uobičajeno. Studija koja je analizirala preko 500 aplikacija je pokazala kako se prosječna aplikacija sastoji od 460 softverskih komponenti, od kojih su 85% komponente otvorenog koda [32]. Izvještaj Synopsya pokazuje da 96% projekata sadrži komponente otvorenog koda [33, str. 4], a prosječna aplikacija ovisi o 298 komponenti otvorenog koda. Korištenjem komponenti otvorenog koda smanjuje se vrijeme i trošak razvoja, ali se pojavljuju novi rizici.

Svaka ovisna komponenta (engl. *dependency*) povećava rizik od ranjivosti. Mnoge komponente otvorenog koda sadrže otkrivene ranjivosti. Prema Synopsysu, oko 60% analiziranih aplikacija ima barem jednu ranjivu komponentu otvorenog koda, dok 40% ima barem jednu komponentu s kritičnom ranjivošću [33, str. 15].

Kompromis lanca dobavljača (engl. *supply chain compromise*) je oblik napada pri kojemu je softver promijenjen u svrhu kompromisa podataka ili sustava pomoću koda maliciozne ovisne komponente, razvojnog ili distribucijskog alata [34].

Neki od načina na koji napadači izvršavaju napade su preuzimanje vlasništva i promjena postojećih komponenti otvorenog koda i stvaranje komponenti s imenom vrlo sličnim postojećoj popularnoj komponenti (engl. *typosquatting*).

Kako bi se smanjio rizik korištenja ranjivih komponenti, organizacija može uvesti politiku o komponentama trećih strana. Politika može uključivati:

- koliko stare smiju biti verzije korištene komponente
- zabranu korištenja neodržavanih komponenti (EOL)
- zabranu korištenja komponenti s poznatim ranjivostima
- popis dozvoljenih i pouzdanih repozitorija
- dozvoljene i nedozvoljene licence
- zahtjeve za ažuriranje komponenti.

Organizacija bi trebala održavati točan popis korištenih komponenti trećih strana (inventar) te uspostaviti plan odgovora na ranjivosti pronađene u njima [9, str. 8]. Održavanje inventara korištenih komponenti, kao i identificiranje poznatih ranjivosti u njima omogućuju alati za analizu kompozicije softvera (SCA, engl. *software composition analysis*). Ovaj popis se još naziva i sastavnica (engl. *bill of materials*).

SCA alati omogućuju izradu izvještaja o svim korištenim komponentama trećih strana u projektu, uključujući i njihove ovisnosti. Izvještaj će također sadržavati podatke o poznatim ranjivostima u korištenim verzijama komponenti.

Primjer SCA alata je *OWASP's Dependency Check* koji omogućuje otkrivanje ovisnosti projekta koje sadrže javno objavljene ranjivosti u nacionalnoj bazi ranjivosti

NIST-a (engl. *National Vulnerability Database*). Postoje i drugi alati otvorenog koda (npr. *Bundler Audit*, *RetireJS*, *SafeNuGet*), kao i komercijalna rješenja (npr. Synopsysov *Black Duck*, *Gemnasium* i *Snyk*).

SCA alati se najčešće koriste kroz integraciju s nekim od *build* (*Maven*, *Ant*, *Gradle*, itd.) ili *CI* alata (npr. *Jenkins*, *Travis*, *CircleCI*), a mogu i poslati obavijest kada se pojave nove ovisnosti kako bi mogle biti ručno pregledane [3, str. 89]. Njima je također moguće provoditi politiku o licencama komponenti trećih strana.

7. Primjena

7.1. Pregled produkcije

Dizajn i razvoj sigurnog softvera nisu dovoljni za sigurnost sustava. Kako bi se smanjio rizik od pojave sigurnosnih propusta prilikom puštanja softvera u primjenu, obavlja se sigurnosni pregled proizvodne okoline sustava. Cilj ovog pregleda je smanjivanje površine napada i osiguravanje odgovarajućeg otvrdnjavanja operacijskog sustava i pojedinih komponenti sustava [10, str. 92].

Kada god je to moguće, pregled se obavlja na stvarnom produkcijskom sustavu ili slikama sustava koje se koriste za instalaciju sustava. U situacijama kada to nije moguće, potrebno je napraviti ispitnu verziju sustava, a nakon pregleda primijeniti nalaze na produkcijski sustav [10, str. 92]. Pregled obavljaju sigurnosni, *DevOps* ili razvojni inženjeri.

Tokom ove aktivnosti pregledavaju se postavke svih *host* računala u sustavu, postavke operacijskih sustava, servisa, aplikacija, kontejnera, itd. Posebni naglasak se stavlja na pregled komponenti trećih strana jer njihov dizajn i arhitektura nisu pod kontrolom organizacije [13, str. 12].

Kao dio otvrdnjavanja sustava, obično se provode sljedeći koraci [3, str. 253]:

- onemogućavanje ili brisanje zadanih i nekorisćenih računa
- promjena zadanih lozinki
- konfiguracija minimalne duljine lozinki, isteka lozinki i sličnih pravila
- onemogućavanje prijavljivanja na sustav kao *root* korisnik
- konfiguracija kontrole pristupa (dozvole pristup datotekama, ACL, MAC)
- uklanjanje instalacija nepotrebnih softverskih paketa
- onemogućavanje servisa i zatvaranje mrežnih portova koji nisu apsolutno potrebni
- isključivanje nesigurnih opcija servisa (npr. nesigurni protokoli i kriptografske postavke)

- ograničavanje pristupa servisa sistemskim pozivima, mreži i uređajima pokretanjem u zaštićenoj okolini (engl. *service sandboxing*)
- ažuriranje softverskih paketa na sigurnim i odobrenim verzijama
- konfiguracija sustava za logiranje aktivnosti korisnika
- konfiguracija sustava za reviziju integriteta datoteka
- konfiguracija vatrozida na domaćinu i konfiguracija ulaznih i izlaznih pravila
- uključivanje potpune enkripcije diska
- isključivanje nekorištenih mrežnih uređaja i konektora (npr. USB, Bluetooth, WiFi).

7.2. Nadziranje i obavještanje

Nakon što je sustav pušten u produkciju, potrebno je nadzirati ponašanje sustava radi traženja znakova napada. Brzo otkrivanje znakova napada je bitno kako bi se pravovremenim odgovorom na incidente napad zaustavio ili ograničio opseg napada, i smanjila potencijalna šteta. Zbog ovoga je potrebno uvesti odgovarajuće mehanizme za nadzor i obavještanje.

Mogući izvori znakova napada u produkcijskom okruženju su:

- IDS, IPS i drugi sustavi za nadzor
- upozorenja SIEM sustava
- antivirusni i antispam softver
- softver za provjeru integriteta podataka i konfiguracija
- logovi operacijskog sustava, aplikacijski logovi i mrežni logovi.

Logiranje je osnova nadzora sigurnosti, jer se može upotrebljavati za otkrivanje znakova napada kao i pogrešaka u izvršavanju aplikacije. Organizacija može politikom definirati zahtjeve za logiranje kako bi se u slučaju incidenta mogla provesti revizija kako bi se utvrdilo što se dogodilo. Logovi moraju omogućiti razumijevanje ponašanje sustava i radnji njegovih korisnika [11, str. 14].

Revizija (engl. *auditing*) znači vođenje evidencije aktivnosti u sustavu. Potrebno je bilježiti tko je radio što i kada. Primjeri događaja bitnih reviziju koje je potrebno logirati [3, str. 79]:

- pristup kritičnim ili osjetljivim podacima
- pristup *root* i administrator korisnicima
- kršenja kontrole pristupa
- autentikacija i pokušaji autentikacije
- svaka promjena autentikacijskih podataka i dodavanje novih korisnika
- sistemski događaji poput pokretanja, gašenja, ponovnog pokretanja i pogrešaka
- pristup revizijskim tragovima
- pogreške validacije podataka
- pogreške za vrijeme izvršavanja koda.

Analizom pogrešaka u izvršavanju aplikacije se mogu pronaći indikatori napada. Na primjer, u web aplikaciji uzastopne greške 400 i 500 mogu biti znak pogrešaka u kodu ili znak skeniranja sustava. SQL pogreške na razini baze podataka mogu biti uzrokovane pogreškama u kodu ili pokušajima SQL umetanja. Pogreške pri pokušaju autentikacije mogu biti pogreška korisnika ili pokušaj neovlaštenog pristupa sustavu [3, str. 262].

U slučajevima kada logovi pokazuju da ponašanje sustava neočekivano odstupa od normalnog, logove treba proučiti i reagirati na odgovarajući način. Korištenjem aplikacija za nadzor logova i praćenje grešaka te SIEM sustava moguće je automatizirati detekciju takvih odstupanja i na vrijeme obavijestiti organizaciju o potencijalnom incidentu.

Pošto logovi mogu sadržavati podatke koji bi mogli pomoći napadaču, treba se pobrinuti da su logovi sigurno spremljeni. Iako su logovi sigurno spremljeni, ne smije se logirati osjetljive podatke poput lozinki, ključeva i tokena [3, str. 264]. Također treba

imati na umu da se u slučaju logiranja osobnih podataka, zakoni vezani uz osobne podatke odnose i na logove [12, str. 95].

7.3. Odgovori na incidente

Kako bi se učinkovito i pravovremeno reagiralo na incidente, potrebno je ustanoviti i dokumentirati proceduru odgovora na incidente.

Dokumentirana procedura odgovora na incidente mora uključivati odgovornosti tima za odgovor na incidente, korake za rješavanje incidenta i upute o obavještanju i eskalaciji. Postojanje procedure odgovora na incidente mogu biti nužne za zadovoljavanje propisa o usklađenosti (npr. PCI DSS).

Tim za odgovor na incidente može uključivati razvojne, operativne, systemske i sigurnosne inženjere, ali i osobe iz marketinga i uprave. Članovi tima moraju biti dostupni 24 sata na dan, razumjeti svoje odgovornosti, i biti u mogućnosti izvršiti potrebne radnje u slučaju incidenta [10, str. 57].

Incidenti mogu biti tehnički ili sigurnosni, i za odgovore na njih mogu biti zaduženi različiti timovi. U slučaju tehničkih incidenata, prioritet je da se sustav vrati u radno stanje u što kraćem vremenu. U slučaju sigurnosnih incidenata, prioriteti mogu biti drugačiji: potrebno je utvrditi opseg napada, zadržati njegovo širenje, napraviti slike sustava i kopije logova za forenzičku analizu [3, str. 271].

Prema preporukama NIST-a odgovor na sigurnosne incidente uključuje sljedeće korake [35, str. 21]:

- Priprema: pružanje potrebnih informacija, dokumentacije, opreme i pristupa timu za odgovore na incidente.
- Detekcija i analiza: utvrđivanje vektora napada, znakova incidenta, izvora preduvjeta za incident, analiza i dokumentacija incidenta, izvješćivanje.
- Zadržavanje, iskorjenjivanje i oporavak.
- Aktivnost nakon incidenta: naučene lekcije, korištenje prikupljenih podataka o incidentu, spremanje dokaza.

7.4. Post mortem

Kada dođe do incidenta, organizira se post mortem sastanak kako bi se razumjelo što se dogodilo i kako spriječiti da se slični problemi ponove. Održavanje post mortem sastanka sa svim uključenim stranama može znatno pomoći u poboljšanju sigurnosnih mjera i samog razvojnog procesa. Ovaj sastanak se treba održati unutar nekoliko dana od incidenta [3, str. 274], [35, str. 38].

Na ovom sastanku potrebno je razgovarati o sljedećim pitanjima [35, str. 38]:

- Što se točno dogodilo i kada?
- Kako je reagirao tim za odgovore na incidente?
- Da li je odgovor na incident bio učinkovit?
- Da li je tim za odgovore na incidente imao sve potrebne informacije?
- Da li su napravljene radnje koje su negativno djelovale na oporavak?
- Kako treba reagirati sljedeći put u slučaju sličnog incidenta?
- Kako spriječiti slične incidente u budućnosti?
- Kako identificirati probleme ranije? Na koje pokazatelje treba obratiti pažnju kako bi slične incidenti brže otkrili u budućnosti?

Potrebno je napraviti agendu sastanka kako bi se osiguralo da se raspravi o svim bitnim stvarima. Agendu je potrebno podijeliti sa svim sudionicima prije sastanka. Tokom sastanka potrebno je voditi zapisnik i dokumentirati glavne zaključke i daljnje radnje.

Važno je pobrinuti se da sastanak uključuje prave ljude. Ne samo da je važno pozvati ljude koji su bili uključeni u incident, već bi trebalo pozvati i one koji bi mogli biti uključeni u budućim sličnim situacijama. Kako je cilj ovog sastanka razmjena informacija i učenje, dobro je uključiti članove drugih timova kako bi i oni bili svjesni naučenih lekcija, te mogli dati svoje prijedloge za poboljšanje [3, str. 248], [35, str. 39].

Svima uključenima treba biti jasno da su ciljevi post mortem sastanka učenje i poboljšanje procesa, a ne okrivljavanje i kritiziranje. Uključeni se moraju osjećati

sigurno kako bi podijelili sve informacije, bili iskreni i transparentni te sagledali sve činjenice objektivno [3, str. 275].

Ključno je pronaći temeljne uzroke incidenta. Za ovo se može koristiti tehnika pod nazivom "5 zašto". Proces započinje od određenog uočenog problema oko koga se svi slažu. Zatim se postavlja pitanje zašto se to dogodilo, a na odgovor se opet pita zašto, i tako dok se ne dođe do temeljnog uzroka. Neki problemi mogu imati više temeljnih uzroka, pa se za svako pitanje može uzeti u obzir više odgovora.

Ovisno o incidentu, priprema za post mortem može uključivati pregled koda. Pregled koda obavljaju razvojni inženjeri kako bi razumjeli što je pošlo po zlu i da mogu identificirati temeljne uzroke incidenta [3, str. 161].

8. Metodologije i radni okviri

8.1. Microsoft Security Development Lifecycle (SDL)

U ranim 2000-tima Microsoft se suočavao s posljedicama brojnih sigurnosnih propusta u svojim proizvodima. U 2001. crvi *Code Red* i *Nimda* su iskoristili propuste na značajnom broju korisničkih sustava. U to vrijeme je Microsoft zbog sigurnosnih problema često bio kritiziran u medijima i reputacija tvrtke je bila ugrožena [16, str. 33].

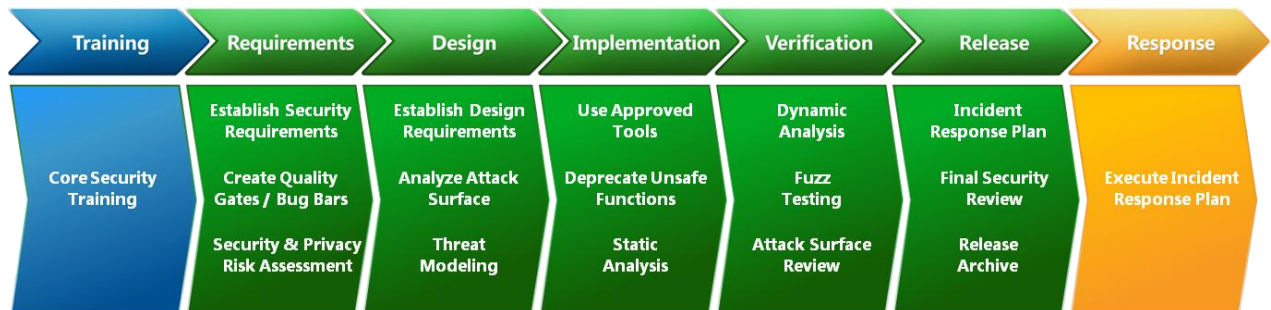
Glavni softverski arhitekt Bill Gates 2002. godine svim zaposlenicima šalje memorandum pod naslovom „Pouzdanost računarstva“ (engl. *Trustworthy Computing*). Gates ističe: „Pouzdanost računarstva je najviši je prioritet svega rada koji obavljamo. Moramo voditi industriju do nove razine pouzdanog računarstva.“ Ovo je bila prekretnica u Microsoftovom odnošenju prema informacijskoj sigurnosti [2, str. 30], [9, str. 3], [36].

Tijekom sljedećih godina Microsoft je promijenio svoj razvojni proces kako bi povećao razinu sigurnosti i privatnosti svojih proizvoda. Novi proces je nazvan *Security Development Lifecycle* (ili SDL). Korištenje SDL procesa u Microsoftu je od 2004. godine nužno za sve proizvode koji su izloženi značajnom riziku ili obrađuju osjetljive podatke. Razvoj i primjena SDL procesa je velika promjena u načinu kako tvrtka dizajnira, razvija i testira softver.

2005. godine u članku „*The Trustworthy Computing Security Development Lifecycle*“ su objavljena zapažanja i rezultati upotrebe SDL procesa. Prema članku, rezultati su ohrabrujući. Operacijski sustav Windows Server 2003 koji je razvijan pomoću SDL procesa je sadržavao 61% manje kritičnih sigurnosnih propusta od operacijskog sustava Windows 2000 koji je razvijan prije uvođenja SDL procesa. Nadalje, znatno je smanjen broj propusta u softverskim paketima SQL Server 2000 i Exchange Server 2000 [37]. Prema članku, modeliranje prijetnji, pregled koda, automatizirani alati za analizu koda i testiranje fuzz metodom su mnogo učinkovitiji i temeljitiji u sprečavanju ili uklanjanju sigurnosnih propusta od klasičnog penetracijskog testiranja.

Ohrabren dobrim rezultatima u internoj upotrebi procesa, Microsoft se zalaže da SDL učini široko dostupnim. 2006. godine je objavljena knjiga „*The Security Development Lifecycle*“ u kojoj su opisane sve faze SDL procesa, najbolje prakse i

upute za inkrementalnu implementaciju procesa u organizaciji [16]. SDL proces se nastavlja razvijati i nadopunjavati. Trenutno je aktualna verzija vodiča 5.2 koja je objavljena 2012. godine [10].



Slika 4: The Security Development Lifecycle (SDL)

SDL uključuje niz praksi usmjerenih na sigurnost u svaku fazu razvojnog procesa softvera [9, str. 5]. Prakse su:

1. Osiguravanje obuke: Inženjeri i rukovoditelji moraju razumjeti osnove sigurnosti i znati kako učiniti proizvode sigurnijima. Cjelokupna organizacija treba biti svjesna sigurnosnih prijetnji i napadačkih tehnika i ciljeva.
2. Definiranje sigurnosnih zahtjeva: Sigurnosne zahtjeve treba definirati tijekom početnih faza projektiranja i planiranja, te ih treba stalno ažurirati u skladu s promjenama zahtjeva funkcionalnosti.
3. Definiranje metrike i izvješćivanje o usklađenosti: Potrebno je definirati minimalne prihvatljive razine sigurnosti kako bi inženjerski timovi razumjeli rizike i mogli odgovarati za ispunjavanje tih kriterija. Sigurnosni nedostaci i sigurnosni zadatci moraju biti jasno označeni kao sigurnosni, te klasificirani odgovarajućom sigurnosnom ozbiljnošću. To omogućava prioritizaciju zadataka i točno praćenje i izvješćavanje o radu na sigurnosti.
4. Modeliranje prijetnji: Učinkovita i jeftina metoda identifikacije sigurnosnih ranjivosti, određivanja rizika, te odabira i uspostave odgovarajućih ublažavanja.

5. Uspostavljanje projektnih zahtjeva: Projektni zahtjevi definiraju sigurnosne značajke koje je potrebno implementirati. Na primjer: kriptografija, autentikacije, autorizacija, evidentiranje, itd.
6. Definiranje i korištenje kriptografskih standarda: Generalno je pravilo da se trebaju koristiti samo provjerene biblioteke koda i algoritmi za šifriranje. Sustavi bi trebali biti dizajnirani tako da je moguće lako zamijeniti metodu šifriranja, ako to postane potrebno.
7. Upravljanje sigurnosnim rizikom upotrebe komponenti treće strane: Prilikom odabira komponenti treće strane potrebno je uzeti u obzir utjecaj na sigurnost. Održavanje popisa komponenti treće strane i plan odgovora u slučaju otkivanja novih ranjivosti su jedna od metoda za ublažavanju ovog rizika.
8. Korištenje odobrenih alata: Organizacija treba objaviti popis odobrenih alata i relevantnih sigurnosnih postavke. Treba nastojati koristiti najnovije verzije odobrenih alata i iskoristiti nove funkcionalnosti sigurnosnih analiza i zaštite.
9. Statička analiza sigurnosti (SAST): Korištenje SAST alata za automatizirani pregled koda kako bi se pronašli sigurnosni propusti i osiguralo provođenje praksi sigurnog kodiranja.
10. Dinamička analiza sigurnosti (DAST): Korištenje DAST alata za testiranje sigurnosti za vrijeme izvršavanja aplikacije.
11. Penetracijsko testiranje: Provodi se kao testiranje „crne kutije“ ili u kombinaciji s automatiziranim i ručnim pregledom koda kako bi se dobila veća razina analize.
12. Uspostavljanje standardnog postupka odgovora na incidente: Plan postupka odgovora na incidente treba uključivati koga kontaktirati u slučaju sigurnosnog incidenta, i definirati protokole za ublažavanje ranjivosti, ispravljanje propusta i komunikaciju s korisnicima. Plan se mora testirati prije nego što bude potreban.

8.2. OWASP SAMM

SAMM (Software Assurance Maturity Model) je radni okvir koji pruža učinkovit i mjerljiv način analize i poboljšanja sigurnosti softvera. *SAMM* organizacijama omogućuje analiziranje i procjenu trenutnih sigurnosnih praksi, te formuliranje i provedbu strategije za sigurnost softvera koja je prilagođena rizicima s kojima se organizacija suočava [9, str. 26], [13, str. 5].

Prvi korak u primjeni ovog modela je izvršavanje samoprocjene da bi se dobio pregled stanja sigurnosnih aktivnosti u organizaciji. Nakon toga slijedi definiranje cilja organizacije i plana provedbe pomoću propisanih savjeta o implementaciji sigurnosnih aktivnosti. Povremenom samoprocjenom se obavlja mjerenje poboljšanja sigurnosti organizacije.

SAMM definira pet kritičnih poslovnih funkcija: upravljanje, dizajn, implementacija, provjera i operacije. Svaka poslovna funkcija je kategorija aktivnosti razvoja softvera. Za svaku poslovnu funkciju, definirane su tri sigurnosne prakse. Svaka sigurnosna praksa je područje sigurnosnih aktivnosti koje izgrađuju sigurnost unutar poslovne funkcije. Sigurnosne aktivnosti unutar svake prakse su podijeljene u dvije grupe s odvojenim ciljevima [13, str. 5], i prema razini zrelosti (između 1 i 3). Aktivnosti koje spadaju u nižu razinu zrelosti je obično lakše implementirati i zahtijevaju manje ulaganja. Organizacija može odabrati željenu razinu zrelosti koju želi postići svakom sigurnosnom praksom, ovisno o njezinim potrebama.

Upravljanje je poslovna funkcija koja se bavi time kako organizacija upravlja cjelokupnim aktivnostima na razvoju softvera, uključujući poslovne procese i aktivnosti na razini organizacije. Sigurnosne prakse definirane unutar ove funkcija su: [13, str. 7–8]

- Strategija i metrike: izgradnja učinkovitog i djelotvornog plana za realizaciju sigurnosnih ciljeva softvera unutar organizacije.
- Politika i usklađenost: poticanje razumijevanja i provođenja unutarnjih i vanjskih standarda i propisa.
- Obrazovanje i vođenje: povećanje znanja i svijesti o sigurnom softveru svih članova organizacije.

Dizajn je poslovna funkcija koja pokriva procese koji se bave definiranjem ciljeva organizacije i načinom izrade softvera. Dizajn uključuje prikupljanje zahtjeva, definiranje arhitekture i dizajna softvera. Sigurnosne prakse ove funkcije su: [13, str. 8–9]

- Procjena prijetnji: Prepoznavanje i klasifikacija rizika projekata temeljenih na funkcionalnosti razvijanog softvera i karakteristikama okruženja.
- Sigurnosni zahtjevi: Definiranje odgovarajućih sigurnosnih zahtjeva softvera u skladu s potrebama zaštite usluge i podataka.
- Arhitektura sigurnosti: Korištenje praksi sigurnog dizajna, analiza sigurnosti arhitekture i korištenih tehnologija.

Implementacija je poslovna funkcija koja uključuje aktivnosti vezane uz samu implementaciju softvera. Sigurnosne prakse ove funkcije su: [13, str. 9–10]

- Sigurna gradnja: Automatizacija i standardizacija softverske izrade. Automatizacija sigurnosnih provjera (npr. *SAST* i *DAST*). Upravljanje rizicima korištenja softverskih ovisnosti.
- Sigurno stavljanje u primjenu: Povećanje sigurnosti primjene softvera u produkcijskom okruženju provjeravanjem integriteta softvera i automatizacija kako bi se smanjila mogućnost pogreške. Zaštita osjetljivih i tajnih podataka poput lozinki, tokena i ključeva.
- Upravljanje defektima: Prikupljanje i analiza podataka o sigurnosnim defektima, te dobivanje metrika za pomoć pri donošenju odluka o sigurnosti.

Provjera obuhvaća aktivnosti povezane s provjeravanjem i testiranjem artefakata proizvedenih tijekom razvoja softvera. Sigurnosne prakse ove funkcije su: [13, str. 10–11]

- Procjena arhitekture: Provjera ispunjavanja zahtjeva utvrđenih u praksi sigurnosnih zahtjeva. Provjera da li arhitektura ublažava tipične prijetnje i specifične prijetnje identificirane u praksi procjene prijetnji.
- Testiranje ispunjavanja zahtjeva: Provjera da implementirane sigurnosne kontrole djeluju prema očekivanjima i udovoljavaju sigurnosnim zahtjevima.

- Sigurnosno testiranje: Korištenje automatiziranih alata za otkrivanje i ispravljanje jednostavnijih problema, kako bi se kod ručnog testiranja mogao staviti veći fokus na složenije vektore napada.

Operacije uključuju aktivnosti koje osiguravaju povjerljivost, integritet i dostupnost usluge i podataka za vrijeme primjene aplikacije. Sigurnosne prakse ove funkcije su: [13, str. 11–12]

- Upravljanje incidentima: Poboljšavanje otkrivanja sigurnosnih incidenata, ograničavanje štete i povratak u normalni rad kroz odgovor na incidente.
- Upravljanje okruženjem: Poboljšavanje i održavanje sigurnosti okruženja aplikacije (otvrdnjavanje sustava, instaliranje zakrpa i ažuriranje paketa).
- Operativno upravljanje: Postavljanje i administracija sustava, stavljanje sustava van pogona, postavljanje i administracija baze podataka, upravljanje sigurnosnim kopijama, itd.

8.3. BSIMM

BSIMM (Building Security In Maturity Model) je radni okvir koji pomaže organizacijama da uvedu sigurnost u proces razvoja softvera i usporede zrelost procesa s drugim organizacijama.

Prvu verziju su 2009. godine razvili Gary McGraw, Sammy Migués i Brian Chess koristeći ranu verziju *SAMM*-a [9, str. 27]. Ideja je bila stvoriti deskriptivni model promatranjem aktivnosti procesa razvoja softvera i korištenjem stvarnih podataka o sigurnosti softvera u različitim tvrtkama. Glavna razlika u odnosu na *SAMM* model je da *BSIMM* ne propisuje kako uvoditi sigurnost u razvoj softvera, već samo izvještava o korištenim praksama.

BSIMM se nastavlja razvijati. Tvrtka Synopsys izdaje novu verziju *BSIMM*-a jednom godišnje. Svaka nova verzija uključuje podatke prikupljene iz više organizacija, pa *BSIMM* predstavlja višegodišnju studiju o trenutnom stanju sigurnosti softvera. Posljednja verzija uključuje podatke iz 122 tvrtke [9, str. 27], [38, str. 32].

BSIMM predstavlja ukupno 119 različitih sigurnosnih aktivnosti iz 12 sigurnosnih praksi, podijeljenih u 4 domene.

Sigurnosne aktivnosti su označene trima razinama. Razine aktivnosti predstavljaju relativnu učestalost s kojima se aktivnosti primjenjuju u organizacijama. Najčešće primjenjivane aktivnosti su označene razinom 1, manje često razinom 2, a rijetko korištene aktivnosti razinom 3 [38, str. 3].

Domena “Upravljanje” sadrži prakse koje pomažu u organizaciji i upravljanju sigurnošću softvera i uključuje sljedeće aktivnosti [38, str. 40–47]:

- Strategija i metrike: Planiranje, podjela uloga i odgovornosti, određivanje sigurnosnih ciljeva softvera, određivanje proračuna, utvrđivanje metrika i prihvatljivih granica.
- Usklađenost i politika: Identificiranje kontrola za provođenje usklađenosti, umanjivanje rizika pomoću ugovornih kontrola (poput SLA), uvođenje i revizije sigurnosnih politika.
- Trening: Provođenje treninga u svrhu podizanja svijesti o sigurnosti i povećanje znanja unutar organizacije.

Domena “Inteligencija” sadrži prakse koje se bave prikupljanjem znanja o sigurnosnim aktivnostima u softveru unutar organizacije i uključuje sljedeće aktivnosti [38, str. 47–52]:

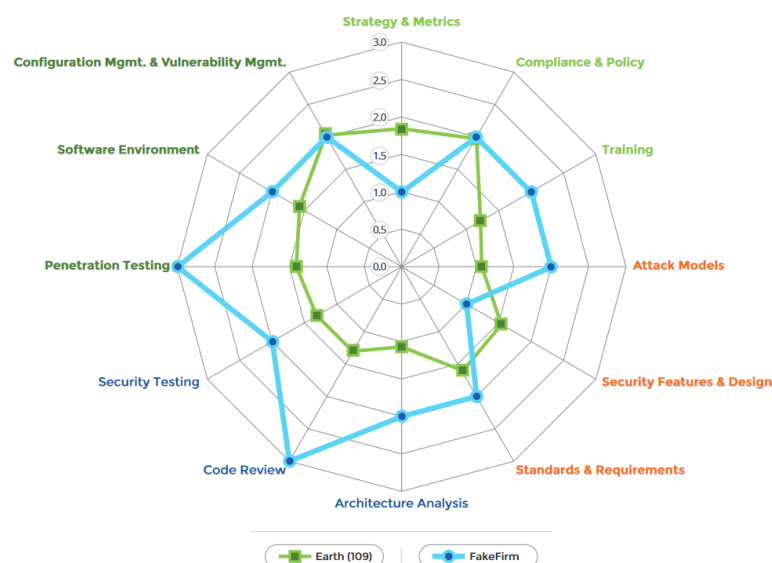
- Modeli napada: Modeliranje prijetnji, identifikacija potencijalnih napadača, razvoj scenarija zloupotrebe, klasifikacija podataka i obrasci napada specifični za tehnologiju.
- Sigurnosne značajke i dizajn: Implementacija sigurnosnih značajki, izrada sigurnih i provjerenih radnih okvira i knjižnica koda, te njihovo korištenje.
- Standardi i zahtjevi: Stvaranje sigurnosnih standarda, održavanje središnjeg mjesta za informacije o sigurnosti softvera, pretvaranje propisa i politika u zahtjeve.

Domena “Dodirne točke razvojnog procesa softvera” sadrži prakse vezane uz analizu i osiguranje pojedinih elemenata procesa razvoja softvera. Domena uključuje sljedeće aktivnosti [38, str. 53–57]:

- Analiza arhitekture: Usvajanje postupaka analize arhitekture i dizajna, pregled sigurnosnih značajki, izrada plana procjene i uklanjanja arhitektonskih rizika.
- Pregled koda: Ad-hoc pregled koda, ručni i automatizirani pregled koda, izrada prilagođenih pravila i profila za automatizirani pregled, centralizirano izvještavanje i praćenje metrika.
- Sigurnosno testiranje: Testiranje sigurnosnih zahtjeva i značajki. Primjena alata za skeniranje ranjivosti i testiranje fuzz metodom. Primjena modela napada i pokrivenosti koda.

Domena “Stavljanje u primjenu” uključuje sljedeće aktivnosti [38, str. 58–62]:

- Penetracijsko testiranje: Identifikacija ranjivosti penetracijskim testiranjem, testiranje crne kutije i bijele kutije.
- Softversko okruženje: Nadgledanje sustava, zaštita i potpisivanje koda, instalacija zakrpa i ažuriranje softverskih paketa, orkestracija, virtualizacija i kontejnerizacija.
- Upravljanje konfiguracijom i upravljanje ranjivostima: Uspostavljanje procedure odgovora na incidente, pronalazak i ispravljanje pogrešaka pomoću podataka iz produkcije (nadgledanje i logiranje).



Slika 5: Rezultati procjene BSIMM prikazani su u obliku grafikona

Tvrtka Synopsys pruža uslugu procjene zrelosti organizacije putem intervjua od strane stručnjaka za sigurnost softvera (Slika 5), i ne pruža alat za samostalnu procjenu zrelosti prema BSIMM modelu [9, str. 28].

9. Kontinuirana integracija

Cilj ovog poglavlja je implementacija procesa kontinuirane integracije (CI, engl. *continuous integration*) koji koristi automatizirane alate za poboljšanje sigurnosti razvojnog procesa softvera.

Kontinuirana integracija je proces automatizacije testiranja i izgradnje koda svaki put kada su promjene koda predane u sustav kontrole verzija (engl. *version control system*). CI proces osigurava automatizirane testove koji stalno provjeravaju da je kod spreman za uporabu [3, str. 19]. CI je jedna od uobičajenih *DevOps* praksi koje spajaju razvoj softvera i IT operacije. Uključivanje sigurnosnih aktivnosti u *DevOps* prakse se naziva *DevSecOps* [39].

Za primjer implementacije CI procesa korišten je projekt otvorenog koda razvijan programskim jezicima Python i JavaScript¹⁰. Kao sustav za kontrolu verzija korišten je servis *GitHub*¹¹. Konfigurirana je integracija s nekoliko servisa koji analiziraju i testiraju kod. Svaki puta kada je predana promjena koda na pregled, GitHub šalje signal integriranim servisima i pokreće se niz automatiziranih provjera. Nakon što završe automatizirane provjere, slijedi ručni pregled koda i rezultata provjera, i odlučuje se da li će kod biti prihvaćen ili ne.

Korištene su sljedeće provjere:

- izvršavanje Python i JavaScript testova te analiza pokrivenosti koda
- analiza Python i JavaScript koda linter alatima
- analiza koda servisa *SonarCloud*¹², *LGTM*¹³ i *Code Climate Quality*¹⁴
- detekcija ranjivih ovisnosti servisom *Snyk*¹⁵.

¹⁰NearBeach: <https://github.com/robotichhead/NearBeach>;
<https://github.com/draganHR/NearBeach-ssdlc>

¹¹ GitHub: <https://github.com/>

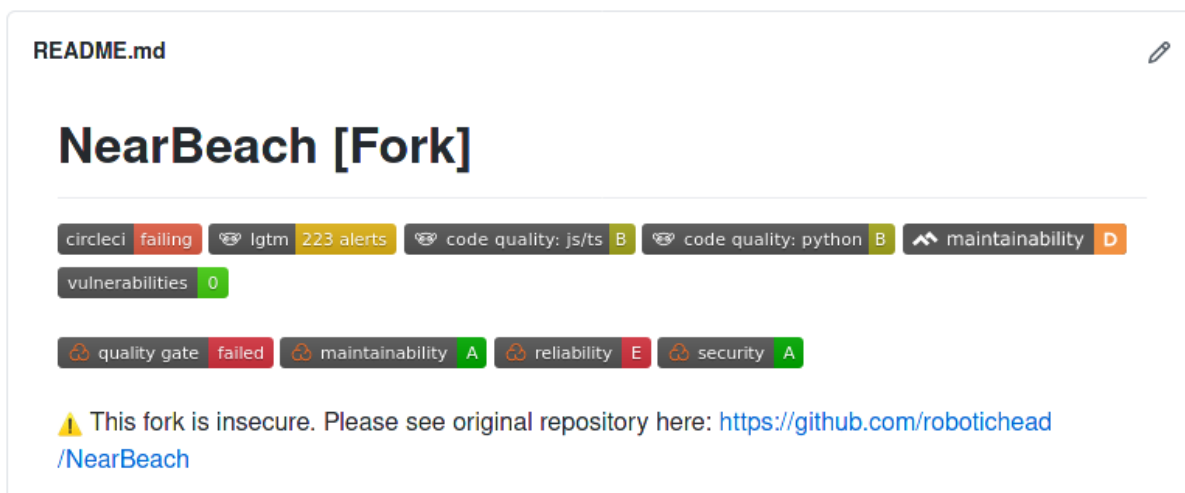
¹² SonarCloud: <https://sonarcloud.io/>

¹³ LGTM: <https://lgtm.com/>

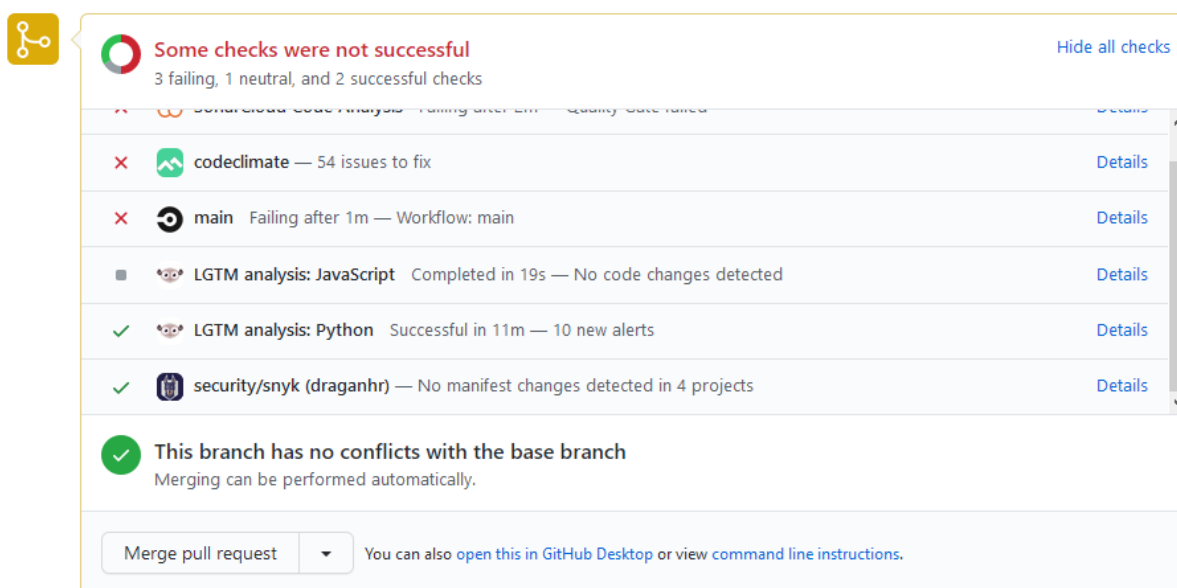
¹⁴ Code Climate Quality: <https://codeclimate.com/>

¹⁵ Snyk: <https://snyk.io/>

Rezultati svih provjera su prikazani pomoću oznaka provjera na glavnoj stranici projekta na *GitHub* servisu (Slika 6).



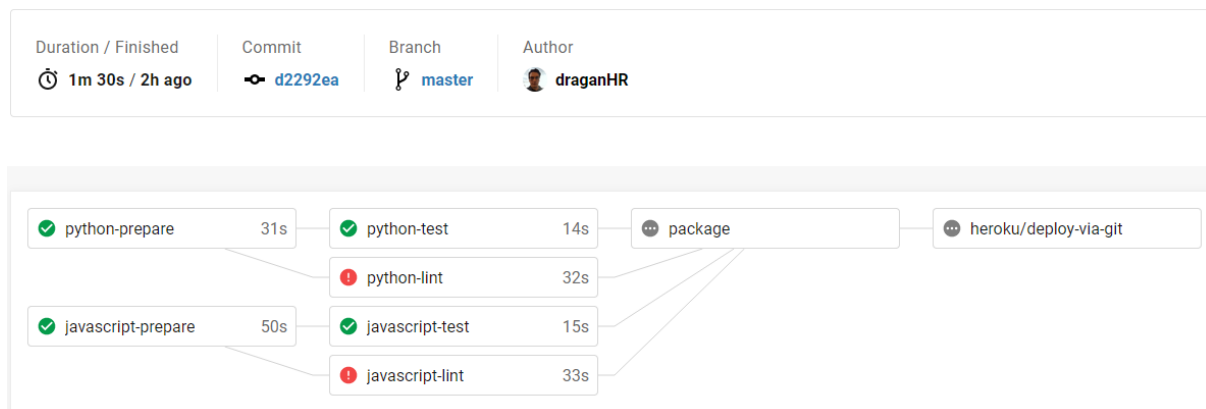
Slika 6: Oznake rezultata provjera procesa kontinuirane integracije



Slika 7: Prikaz rezultata provjera promjena koda

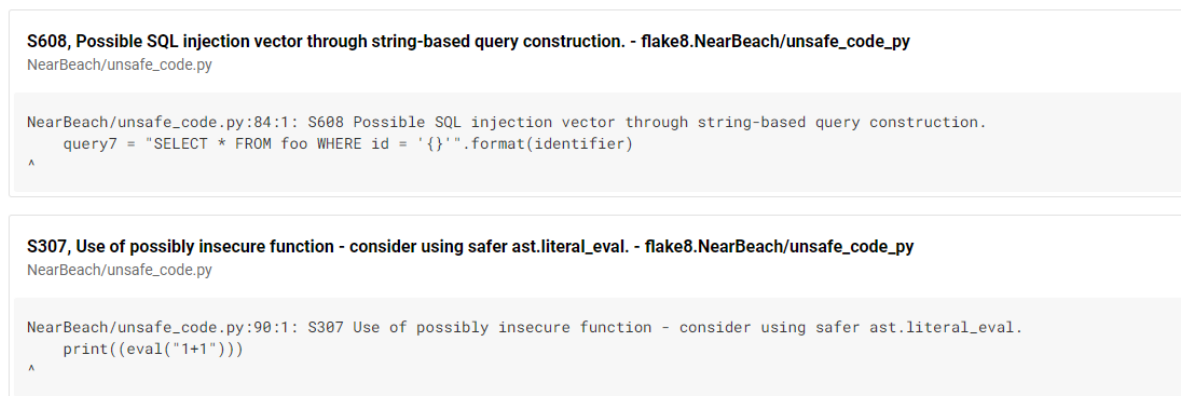
Rezultati provjera za predane promjene koda su vidljive na stranici za pregled promjene koda (Slika 7). U slučaju da provjera koda završi neuspješno, autor koda će primiti obavijest putem e-mail poruke. *GitHub* omogućuje uključivanja zaštite koda tako da se odaberu provjere koje moraju završiti uspješno, u protivnome kod ne može biti prihvaćen.

Izvršavanje Python i JavaScript testova i linter alatima se izvršava pomoću servisa za izvršavanje cjevovoda kontinuirane integracije *CircleCI*¹⁶. Konfiguracijska datoteka definira korake cjevovoda, to jest kako pripremiti okruženje, kako izvršiti testove i analizirati njihove rezultate (Slika 8).



Slika 8: Cjevovod kontinuirane integracije


Neki linteri imaju mogućnost identifikacije nekih sigurnosnih propusta ili korištenja nesigurnih praksi kodiranja. U ovom primjeru, korišten je Python linter *Flake8* zajedno s paketom *Bandit*, i uspio je detektirati niz sigurnosnih propusta, uključujući korištenje nesigurne serijalizacije, umetanja naredbi i SQL umetanja (Slika 9).



Slika 9: Sigurnosni propusti detektirani korištenjem alata *Bandit*

¹⁶ CircleCI: <https://circleci.com/>

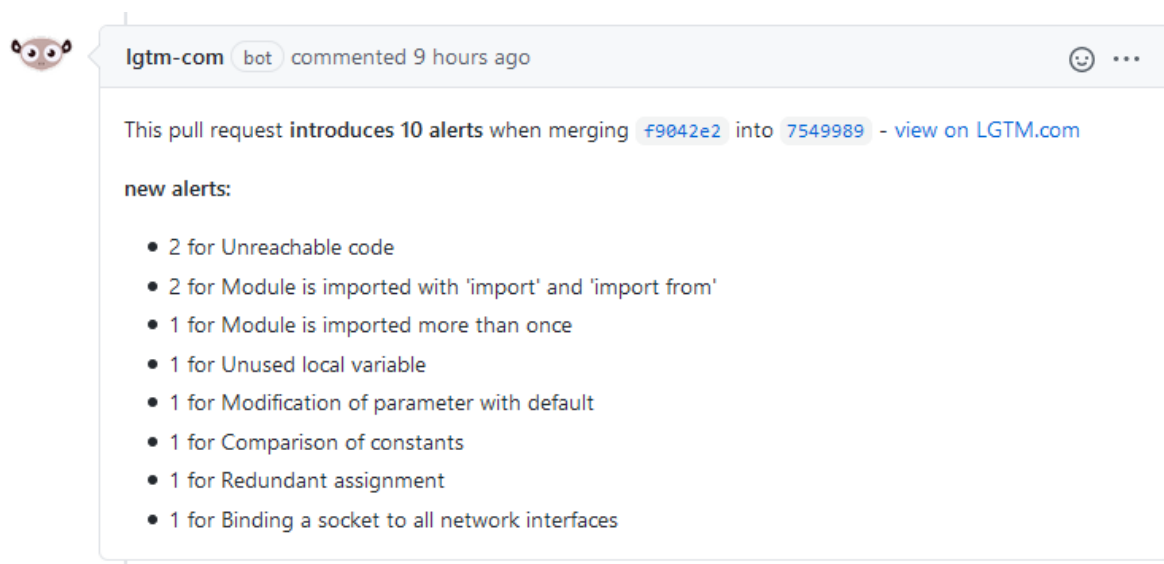
Rezultatima izvršavanja testova i linter alata moguće je pristupiti na *CircleCI* servisu, kao i izvještaju o pokrivenosti koda u HTML formatu (Slika 10).

Coverage report: 83% 				
Module ↑	statements	missing	excluded	coverage
NearBeach/forms_special_fields.py	106	106	0	0%
<u>NearBeach/models.py</u>	1093	54	0	95%
NearBeach/templates/__init__.py	0	0	0	100%
NearBeach/templatetags/__init__.py	0	0	0	100%
NearBeach/templatetags/nearbeach_extras.py	35	35	0	0%
NearBeach/tests/test_examples.py	6	0	0	100%
NearBeach/urls.py	8	8	0	0%
NearBeach/views_lookup.py	14	14	0	0%
Total	1262	217	0	83%

coverage.py v5.2.1, created at 2020-07-26 08:44 +0000

Slika 10: Izvještaj o pokrivenosti koda

Servis *LGTM* nudi uslugu statičke analize i automatiziranog pregleda koda, te može otkriti brojne sigurnosne propuste i ocijeniti kvalitetu koda. Ukoliko ovaj alat pronade propuste u predanim promjenama koda, ostaviti će komentar na stranici za pregled promjena koda (Slika 11).



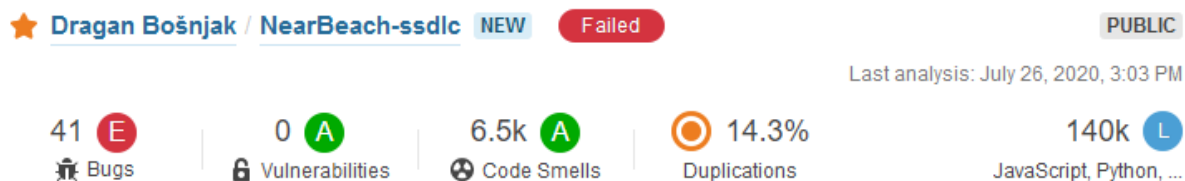
Slika 11: Komentar servisa LGTM na stranici za pregled promjena koda

LGTM je na korištenom projektu uspio pronaći 223 problema, od kojih je 8 svrstao u kategoriju sigurnosnih propusta. Neki od pronađenih problema su primjeri lošeg kodiranja i nedovoljnog poznavanja korištenog programskog jezika. Neki od pronađenih bugova, iako nisu kategorizirani kao sigurnosni propusti mogu dovesti do sigurnosnih problema (Slika 12).



Slika 12: Propusti detektirani servisom LGTM

Servis *SonarCloud* nudi uslugu statičke analize koda za 25 različitih programskih jezika. *SonarCloud* omogućuje neprekidno osiguravanje održavanja, pouzdanosti i sigurnosti koda. Radi se o SAAS varijanti aplikacije *SonarQube*. Analizom koda mogu se pronaći problemi koji su kategorizirani kao „code smell“, „bug“ ili ranjivost, i sigurnosna žarišta, to jest kod koji je zbog potencijalnog utjecaja na sigurnost potrebno ručno pregledati. *SonarCloud* ocjenjuje pouzdanost, sigurnost i održivost analiziranog koda (Slika 13).



Slika 13: Prikaz rezultata skeniranja servisom SonarCloud

Moguće je definirati granice kvalitete (engl. *quality gates*) za preko 20 metrika (Slika 14).

Conditions ⓘ Add Condition

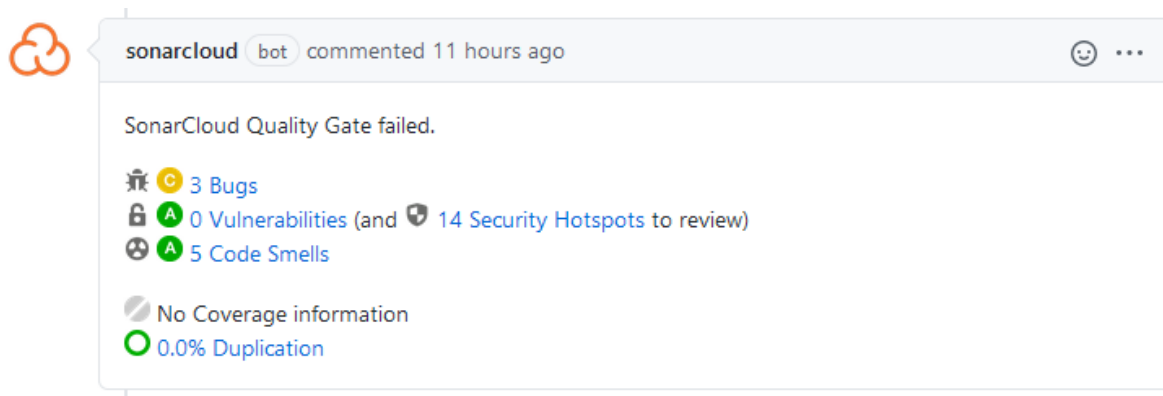
Conditions on New Code

Conditions on New Code apply to all branches and to Pull Requests.

Metric	Operator	Value	Edit	Delete
Coverage	is less than	80.0%		
Duplicated Lines (%)	is greater than	3.0%		
Maintainability Rating	is worse than	A		
Code Smells	is greater than	10		
Vulnerabilities	is greater than	1		
Reliability Rating	is worse than	A		
Security Rating	is worse than	A		

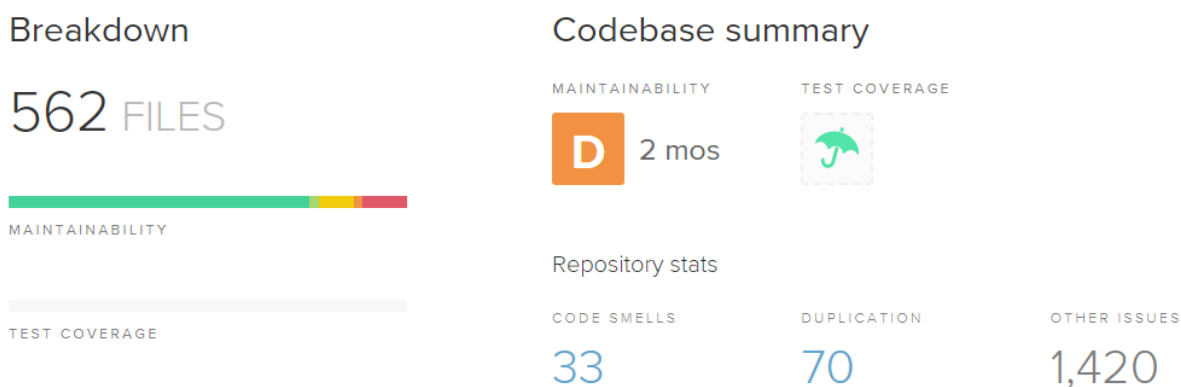
Slika 14: Postavke granica kvalitete u servisu SonarCloud

Kao i *LGTM*, *SonarCloud* će ostaviti komentar na stranici za pregled promjena koda kada otkrije propuste u predanim promjenama koda (Slika 15).



Slika 15: Komentar servisa SonarCloud na stranici za pregled promjena koda

Code Climate Quality je servis koji pruža uslugu automatiziranog pregleda koda za preko 10 programskih jezika. Ovaj servis pri analizi koristi integraciju s preko 30 različitih lintera i alata za statičku analizu, uključujući *Bandit* i *ESLint* koji su već prethodno korišteni u CI cjevovodu, te *SonarPython* koji koristi i *SonarCloud* servis. Detektirani problemi su svrstani prema ozbiljnosti i kategoriji. *Code Climate Quality* ocjenjuje tehnički dug projekta i daje procjenu vremena potrebnog da se isprave nedostaci u kodu (Slika 16).



Slika 16: Rezultat analize koda servisom Code Climate Quality

Ovaj servis također može ostaviti komentar na stranici za pregled promjena koda kada otkrije propuste u predanim promjenama koda.

Code Climate Quality je detektirao više problema od *LGTM-a* i *SonarClouda*, ali većina se odnosi na kršenje pravila stila kodiranja.

Snyk je SCA servis koji pruža uslugu analize ovisnosti i otkrivanja ovisnosti s poznatim ranjivostima za 10 programskih jezika. Analiziraju se ovisnosti koje projekt već koristi kao i promjene ovisnosti u predanim promjenama koda. Servis uspoređuje ovisnosti korištene u analiziranom projektu s vlastitom bazom podataka poznatih ranjivosti. Analiza se izvršava periodički čak i kada nema promjena koda kako bi se primijenili najnoviji podaci o poznatim ranjivostima. Za svaku ranjivu ovisnost Snyk prikazuje detalje o ranjivosti i načinima iskorištavanja, te navodi u kojoj verziji ovisnosti je ranjivost uklonjena (Slika 17).

HIGH SEVERITY

SQL Injection

Affected module: django@3.0

Introduced through: django@3.0, django-select2@7.4.2

Exploit maturity: No known exploit

Fixed in: django@3.0.3, django@2.2.10, django@1.11.28

Detailed paths

- **Introduced through:** django@3.0
- **Introduced through:** django-select2@7.4.2 › django@3.0
- **Introduced through:** django-select2@7.4.2 › django-appconf@1.0.4 › django@3.0

Overview

django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

Affected versions of this package are vulnerable to SQL Injection. If untrusted data is used as a `StringAgg` delimiter (e.g., in Django applications that offer downloads of data as a series of rows with a user-specified column delimiter) by passing a suitably crafted delimiter to a `contrib.postgres.aggregates.StringAgg` instance, it is possible to break escaping and inject malicious SQL.

Slika 17: Prikaz detalja o ranjivosti ovisnosti otkrivene Snykom

10. Zaključak

Siguran proces razvoja softvera nije panacea jer njegov rezultat nije proizvodnja potpuno sigurnog softvera. Nikada nije moguće s potpunom sigurnošću reći da su sve ranjivosti uklonjene iz softvera. Usprkos tome, usvajanje sigurnosti u proces razvoja softvera ima značajnu korist:

- poboljšanje kvalitete i sigurnosti softvera
- smanjivanje troškova ranim pronalaskom i ispravljanjem nedostataka
- smanjivanje poslovnih rizika organizacije
- usklađenost s propisima i standardima.

Primjenom ovakvog procesa sigurnost softvera je pod kontrolom razvojnog tima, umjesto da ovisi o drugom timu nakon što razvoj završi. Kao rezultat, razvojni tim dobiva osjećaj vlasništva i odgovornosti nad cjelokupnom kvalitetom svojih aplikacija [39].

Iako organizacije koriste različite procese razvoja softvera, moguće je primijeniti prakse sigurnog razvoja koje definiraju radni okviri poput SDL, SAMM i BSIMM. Njih je moguće primijeniti na inkrementalan način, to jest postepeno uvoditi sigurnosne aktivnosti u postojeći razvojni proces. Koje aktivnosti će organizacija usvojiti ovisiti će o potrebama i mogućnostima organizacije.

Kako bi siguran proces razvoja softvera bio uspješno primijenjen, potrebno je educirati inženjere o osnovama sigurnosti, sigurnom dizajnu i sigurnom kodiranju. Također je bitno podići svijest u vezi prijetnji, najčešćih ranjivosti softvera i uobičajenih obrazaca napada.

Važno je odabrati odgovarajuće alate. Odabir krivih alata može organizaciju koštati vrijeme i novac, bez da su dovoljno korisni. Treba obratiti pažnju na to koliko su komplicirani za integriranje, održavanje, te da li daju razumljive rezultate. Automatizirani alati poput SAST i skenera ranjivosti često imaju veliki udio *false positive* rezultata. Oslanjati se na rezultate neodgovarajućih alata može stvoriti privid sigurnosti.

Literatura

- [1] D. Bošnjak, „Penetracijsko testiranje web aplikacija“. sij. 2019.
- [2] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.
- [3] L. Bell, Brunton-Spall, R. Smith, i J. Bird, *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline*. O'Reilly Media, 2017.
- [4] „OWASP Secure Coding Practices Quick Reference Guide, Version 2.0“. OWASP, stu. 2010, [Na internetu]. Dostupno na: https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf.
- [5] *Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, 2002.
- [6] A. Jøsang, M. Ødegaard, i E. Oftedal, „Cybersecurity Through Secure Software Development“, 2015.
- [7] K. Klarin, *Programsko inženjerstvo - Skripta*. Sveučilište u Splitu, 2012.
- [8] R. K. Wysocki, *Effective Software Project Management*, 1. izd. Wiley, 2006.
- [9] L. Williams, „Secure Software Lifecycle Knowledge Area“, izd. 1.0, 2019.
- [10] *Microsoft Security Development Lifecycle (SDL) Process Guidance*, 5.2. Microsoft, 2012.
- [11] „Fundamental Practices for Secure Software Development: Essential Elements of a Secure Development Life Cycle Program (Third Edition)“. SAFECode, ožu. 2018, Pristupljeno: svi. 10, 2020. [Na internetu]. Dostupno na: https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf.
- [12] „OWASP Web Security Testing Guide 4.1“. Pristupljeno: svi. 13, 2020. [Na internetu]. Dostupno na: <https://github.com/OWASP/wstg/releases/download/v4.1/wstg-v4.1.pdf>.

- [13] „OWASP SAMM v2.0“. OWASP, Pristupljeno: svi. 10, 2020. [Na internetu]. Dostupno na: <https://github.com/OWASP/samm/blob/master/Supporting%20Resources/v2.0/OWASP-SAMM-v2.0.pdf>.
- [14] J. Boote, „What are software security requirements?“, lip. 06, 2016. <https://www.synopsys.com/blogs/software-security/software-security-requirements/> (pristupljeno svi. 11, 2020).
- [15] „Application Security Verification Standard 4.0“. OWASP, ožu. 2019, Pristupljeno: svi. 14, 2020. [Na internetu]. Dostupno na: https://owasp.org/www-pdf-archive/OWASP_Application_Security_Verification_Standard_4.0-en.pdf.
- [16] M. Howard i S. Lipner, *The Security Development Lifecycle: SDL, a Process for Developing Demonstrably More Secure Software*, 1. izd. Microsoft Press, 2006.
- [17] „Understanding Red Hat security ratings“. <https://access.redhat.com/security/updates/classification> (pristupljeno svi. 11, 2020).
- [18] „Microsoft Vulnerability Severity Classification for Windows“. Microsoft, ruj. 06, 2018, [Na internetu]. Dostupno na: <https://aka.ms/windowsbugbar>.
- [19] S. Barnum i M. Gegick, „Design Principles“. Cigital, Inc., ruj. 19, 2005.
- [20] „OWASP Developer Guide Reboot: Secure Design Principles“. <https://github.com/OWASP/DevGuide/blob/master/02-Design/01-Principles%20of%20Security%20Engineering.md> (pristupljeno svi. 17, 2020).
- [21] R. E. Smith, „A Contemporary Look at Saltzer and Schroeder’s 1975 Design Principles“, *Secur. Priv. IEEE*, stu. 2012, doi: 10.1109/MSP.2012.85.
- [22] J. H. Saltzer i M. D. Schroeder, „The protection of information in computer systems“, *Proc. IEEE*, sv. 63, izd. 9, str. 1278–1308, 1975.
- [23] D. A. Wheeler, „How to Prevent the next Heartbleed“, sij. 29, 2017. <https://dwheeler.com/essays/heartbleed.html> (pristupljeno svi. 09, 2020).

- [24] M. Bland, „Goto Fail, Heartbleed, and Unit Testing Culture“, lip. 03, 2014.
<https://martinfowler.com/articles/testing-culture.html> (pristupljeno svi. 09, 2020).
- [25] L. Williams, G. Kudrjavets, i N. Nagappan, „On the Effectiveness of Unit Test Automation at Microsoft“. North Carolina State University, [Na internetu]. Dostupno na:
https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf.
- [26] *OWASP Code Review Guide*, V2.0. The OWASP Foundation, 2017.
- [27] „Essential Software Security Training for the Microsoft SDL“. Microsoft Corporation, lip. 02, 2010, [Na internetu]. Dostupno na:
<https://www.microsoft.com/en-us/download/details.aspx?id=9950>.
- [28] K. Scarfone, M. Souppaya, A. Cody, i A. Orebaugh, „Technical Guide to Information Security Testing and Assessment (NIST SP 800-115)“. ruj. 30, 2008, Pristupljeno: svi. 13, 2020. [Na internetu]. Dostupno na:
<https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf>.
- [29] A. Harper i D. Regalado, *Gray Hat Hacking The Ethical Hacker's Handbook*. McGraw-Hill, 2018.
- [30] *PCI Data Security Standard (PCI DSS): Information Supplement: Penetration Testing Guidance*, 1.0. 2015.
- [31] „Testiranje Fuzz metodom (NCERT-PUBDOC-2011-01-322_0)“. Nacionalni CERT, [Na internetu]. Dostupno na: https://www.cert.hr/wp-content/uploads/2011/01/NCERT-PUBDOC-2011-01-322_0.pdf.
- [32] „2019 State of the Software Supply Chain Report“. Sonatype Inc., 2019, [Na internetu]. Dostupno na: <https://www.sonatype.com/en-us/software-supply-chain-2019>.
- [33] „2019 Open Source Security and Risk Analysis (OSSRA) Report“. Synopsys, 2019, [Na internetu]. Dostupno na: <https://www.synopsys.com/software-integrity/resources/analyst-reports/2019-open-source-security-risk-analysis.html>.
- [34] „Supply Chain Compromise“, *mitre.org*, svi. 06, 2019.
<https://attack.mitre.org/techniques/T1195/>.

- [35] P. Cichonski, T. Millar, T. Grance, i K. Scarfone, „Computer Security Incident Handling Guide (SP 800-61 Rev. 2)“. NIST, kol. 2012, [Na internetu]. Dostupno na: <https://csrc.nist.gov/publications/detail/sp/800-61/rev-2/final>.
- [36] B. Gates, „Trustworthy Computing“, srp. 17, 2002. <https://www.wired.com/2002/01/bill-gates-trustworthy-computing/> (pristupljeno srp. 17, 2020).
- [37] S. Lipner i M. Howard, „The Trustworthy Computing Security Development Lifecycle“, ožu. 2005. [https://docs.microsoft.com/en-us/previous-versions/ms995349\(v=msdn.10\)](https://docs.microsoft.com/en-us/previous-versions/ms995349(v=msdn.10)) (pristupljeno srp. 17, 2020).
- [38] S. Migués, J. Steven, i M. Ware, „Building Security In Maturity Model (BSIMM) 10“. ruj. 2019.
- [39] „Secure Software Development Lifecycle (SSDLC)“. <https://snyk.io/learn/secure-sdlc/> (pristupljeno srp. 25, 2020).

Summary

The paper discusses improvement of the software security from a development team perspective. It describes activities that can be used to produce safer and more reliable software, primarily by reducing the number of created vulnerabilities, and enabling early vulnerability detection. Activities are covering all software development life cycle phases, including defining requirements, design, implementation, testing, and deployment.

The most commonly used methodologies and frameworks that help advancing security into the software development process are examined. Their features and method of application are described.

Finally, paper shows an example of the implementation of the continuous integration process which implements some of the discussed security activities for vulnerability prevention and detection by using automated tools. The example demonstrates how to effectively improve software development without the need for major investment and significant changes to the established development process.

Keywords: security, software, process, development, testing.

