

Střední průmyslová škola Třebíč

Maturitní práce

Nasazení a hodnocení nástrojů pro bezpečnostní analýzu zdrojového kódu

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2025/2026 Simona Havelková

Zadání práce

Cílem maturitní práce je nasadit a ověřit sadu nástrojů pro statickou analýzu za účelem hledání zranitelností ve zdrojovém kódu, tzv. SAST (Static Application Security Testing). Vybrané SAST nástroje budou ověřeny nad zranitelným kódem webové aplikace vyvinuté v programovacím jazyce JavaScript. Výstupem práce bude analýza a hodnocení SAST nástrojů dle předem určených metrik (přesnost, pokrytí, míra falešně pozitivních nálezů atd.). Práce se zaměří také na úspěšnost SAST v identifikaci zranitelností určitého typu (např. injekční útoky), na základě čehož uvede doporučení pro jejich konfiguraci za účelem redukce falešně pozitivních hlášení.

ABSTRAKT

Účelem této maturitní práce je prozkoumat a vyhodnotit nástroje pro statickou analýzu zdrojového kódu podle definovaných hodnotících ukazatelů (přesnost, pokrytí, míra falešně pozitivních nálezů atd.) nad zranitelným kódem webové aplikace naprogramované v jazyce JavaScript. Teoretická část obsahuje přehled vývoje software od vysvětlení pojmu životního cyklu vývoje software, rozebrání problematiky softwarových zranitelností až po bezpečnostní analýzu zdrojového kódu. V praktické části je proveden výběr technologií, které se následně přichystaly pro jejich použití. Poté začalo testování vybraných SAST nástrojů. Získané výsledky jsou následně vyhodnoceny. Závěrečná část práce obsahuje shrnutí získaných výsledků analýz. Práce tak poskytuje komplexní přehled možností a omezení vybraných SAST nástrojů a jejich přínosů ke zvýšení bezpečnosti software.

KLÍČOVÁ SLOVA

Statická analýza; bezpečný vývoj; softwarové zranitelnosti; SAST nástroje

ABSTRACT

The aim of this graduation thesis is to explore and evaluate tools for static source code analysis based on defined evaluation criteria (such as accuracy, coverage, and the rate of false positives) using a vulnerable web application written in JavaScript. The theoretical part provides an overview of software development, starting with an explanation of the software development life cycle, discussing the issue of software vulnerabilities, and concluding with the security analysis of source code. In the practical part, suitable technologies were selected and prepared for use. Afterwards, testing of the chosen SAST tools was conducted, and the obtained results were evaluated. The final section summarizes the findings of the analyses. The thesis thus provides a comprehensive overview of the capabilities and limitations of the selected SAST tools and their contribution to improving software security.

KEYWORDS

Static analysis; secure development; software vulnerabilities; SAST tools; software development life cycle

PODĚKOVÁNÍ

Děkuji Mgr. Petru Novotnému za cenné připomínky a rady, které mi poskytl při vypracování maturitní práce.

V Třebíči dne 5. února 2026 podpis autora

PROHLÁŠENÍ

Prohlašuji, že jsem tuto práci vypracoval/a samostatně a uvedl/a v ní všechny prameny, literaturu a ostatní zdroje, které jsem použil/a.

V Třebíči dne 5. února 2026

podpis autora

Obsah

[Úvod 8](#_Toc221184901)

[1 Teoretická část 9](#_Toc221184902)

[1.1 Životní cyklus vývoje software 9](#_Toc221184903)

[1.1.1 Modely SDLC 10](#_Toc221184904)

[1.1.2 Verzování vývoje 17](#_Toc221184905)

[1.1.3 Proces DevOps 21](#_Toc221184906)

[1.1.4 Bezpečný vývoj 25](#_Toc221184907)

[1.2 Softwarové zranitelnosti 28](#_Toc221184908)

[1.2.1 Známe zranitelnosti (CVE) 29](#_Toc221184909)

[1.2.2 Kategorizace zranitelností (CWE) 31](#_Toc221184910)

[1.2.3 Hodnocení rizik (CVSS) 32](#_Toc221184911)

[1.3 Bezpečnostní analýza zdrojového kódu 33](#_Toc221184912)

[1.3.1 Statická analýza 34](#_Toc221184913)

[1.3.2 SAST 40](#_Toc221184914)

[1.3.3 Srovnání vybraných nástrojů 42](#_Toc221184915)

[2 Praktická část 43](#_Toc221184916)

[2.1 Výběr technologií 43](#_Toc221184917)

[2.2 Příprava prostředí 44](#_Toc221184918)

[2.3 Testování SAST nástrojů 44](#_Toc221184919)

[2.3.1 GitLab SAST 44](#_Toc221184920)

[2.3.2 Snyk 48](#_Toc221184921)

[2.3.3 Aikido 56](#_Toc221184922)

[2.3.4 SonarQube 72](#_Toc221184923)

[2.3.5 CodeQL 87](#_Toc221184924)

[2.3.6 Horusec 90](#_Toc221184925)

[2.3.7 Bearer 92](#_Toc221184926)

[2.4 Výsledky a diskuze 96](#_Toc221184927)

[3 Závěr 97](#_Toc221184928)

[Seznam použitých zdrojů 98](#_Toc221184929)

[Seznam obrázků 106](#_Toc221184930)

[Seznam tabulek 109](#_Toc221184931)

Úvod

V dnešní době jsou kybernetické útoky mnohem častější než útoky fyzické, a právě proto je nezbytné jim předcházet. Jednou z příčin může být veliké množství zranitelností ve zdrojovém kódu. Jedním z účinných opatření je bezpečný vývoj software, který pomáhá minimalizovat riziko zneužití zranitelností. K dispozici je řada nástrojů a metod, které tento proces podporují – mezi nimi například statickou analýzu zdrojového kódu.

Hlavním cílem této práce je porovnat různé nástroje pro statickou analýzu zdrojového kódu, a to z hlediska jejich vlastností, jako je podpora programovacích jazyků, rychlost odezvy či možnosti integrace do vývojového procesu.

Práce je rozdělena do několika částí. V teoretické části je popsán životní cyklus vývoje software spolu s procesy, které s ním úzce souvisejí – verzování vývoje, DevOps   
a zásady bezpečného programování. V další části jsou vysvětleny softwarové zranitelnosti, jejich význam a představuje související klasifikační a hodnotící systémy CVE (Common Vulnerabilities and Exposures) – seznam veřejně známých zranitelností a ohrožení informační bezpečnosti, CWE (Common Weakness Enumeration) – „slovník“ nebo katalog chyb, které se v softwaru a hardwaru opakují a CVSS (Common Vulnerability Scoring System) – otevřený standard určený pro hodnocení a kvantifikaci závažnosti softwarových zranitelností. Závěr teoretické části se zaměřuje na problematiku bezpečnostní analýzy zdrojového kódu, obecný princip statické analýzy, fungování SAST nástrojů, a nakonec i na srovnání vybraných SAST nástrojů z pohledu jejich vlastností a praktického využití.

# Teoretická část

Teoretická část se zaměřuje na bezpečný vývoj software prevenci jeho zranitelností. Úvodem se zaměřuje na životní cyklus vývoje, verzování, DevOps a principy bezpečného kódu. Dále se věnuje softwarovým zranitelnostem, kategorizaci a hodnocení. Závěrem se zaměřuje na bezpečnostní analýzu zdrojového kódu, statickou analýzu a srovnání vybraných SAST nástrojů pro odhalování chyb a rizik.

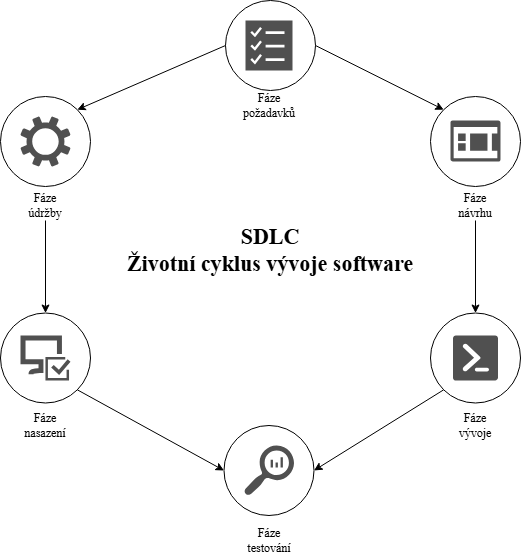
## Životní cyklus vývoje software

Životní cyklus vývoje software, tzv. SDLC (Software Development Life Cycle) je strukturovaný a systematický proces, který vede vývoj software od jeho návrhu až po nasazení a údržbu. SDLC poskytuje jasný rámec pro plánování, tvorbu a správu aplikací, a zajišťuje, že vývoj probíhá efektivně a v souladu s požadavky projektu   
a očekáváními uživatelů [1].

Proces SDLC se obvykle dělí do několika fází [1]:

1. **Plánování** – Identifikovat rozsah, cíl a požadavky projektu
2. **Analýza** – Shromáždit a zkontrolovat data o požadavky projektu
3. **Návrh** – Definovat architektury projektu
4. **Kódování** – Napsat počáteční kód
5. **Testování** – Otestovat kód a odstranit chyby
6. **Nasazení** – Nasadit kód do produkčního prostředí
7. **Údržba** – Průběžně provádět opravy a vylepšení

Tento proces je znázorněn na obrázku 1.



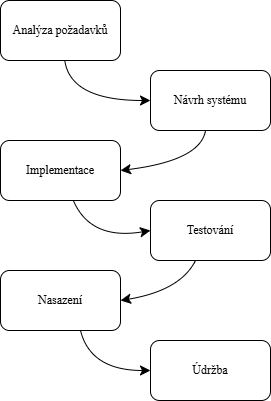
Obrázek 1 Proces SDLC

### Modely SDLC

SDLC má i mnoho modelů:

**Vodopád**

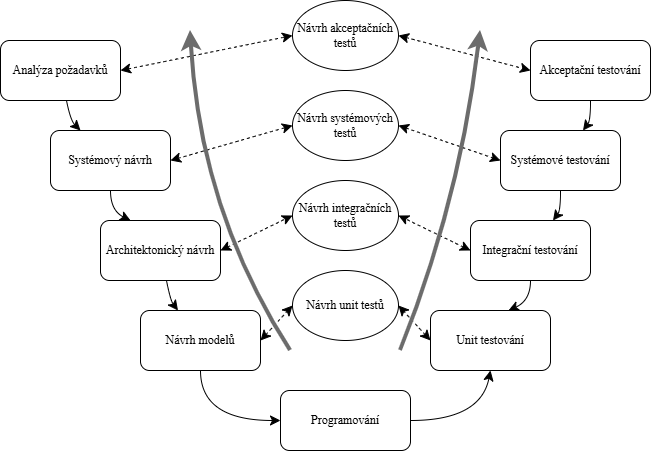
Tento model byl prvním procesním modelem. Označuje se jako lineárně-sekvenční model životního cyklu. Každá fáze musí být dokončena před zahájením další fáze. V modelu Vodopád se fáze nepřekrývají [2]. Obrázek 2 znázorňuje průběh tohoto modelu.



Obrázek 2 Vodopádový model

**V-model**

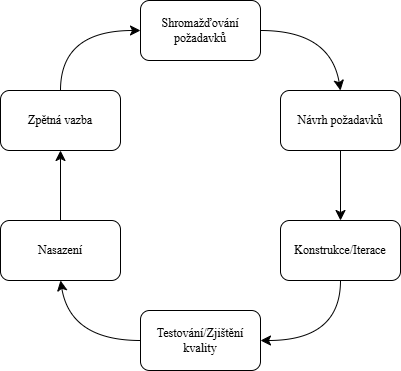
Dalším modelem SDLC je V-model, kde provádění procesů probíhá sekvenčně ve tvaru písmene V. Tento model je rozšířením modelu vodopádu a je založen na propojení testovací fáze s každou odpovídající fází vývoje. Jedná se o vysoce disciplinovaný model a další fáze začíná až po dokončení předchozí fáze [3]. Průběh je zobrazen na obrázku 3.



Obrázek 3 V-model

**Agilní model**

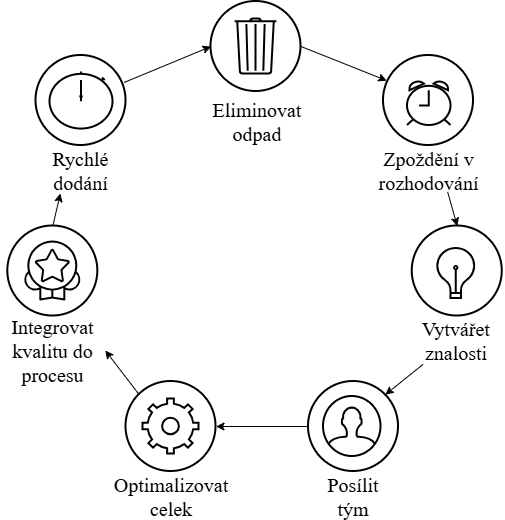
Tento model je velmi odlišný od předchozích modelů. Pomocí tohoto modelu je možné rozdělit projekt do několika dynamických fází, běžně známých jako sprinty. Po každém sprintu týmy reflexují a ohlížejí se zpět, aby zjistily, zda by se dalo něco vylepšit, aby mohly případně upravit svou strategii pro další sprint [4]. Agilní model je vyobrazen na obrázku 4.



Obrázek 4 Agilní model

**Lean**

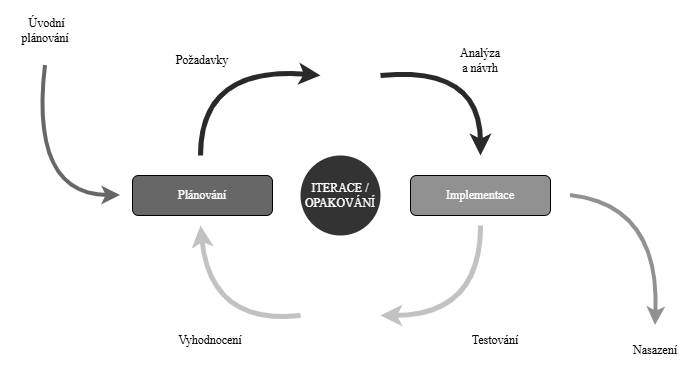
Jak je zobrazené na obrázku 05, Lean model je založený na optimalizaci času a zdrojů vývoje, eliminaci plýtvání a v konečném důsledku dodávání pouze toho, co produkt potřebuje [5].



Obrázek 5 Lean model

**Iterativní model**

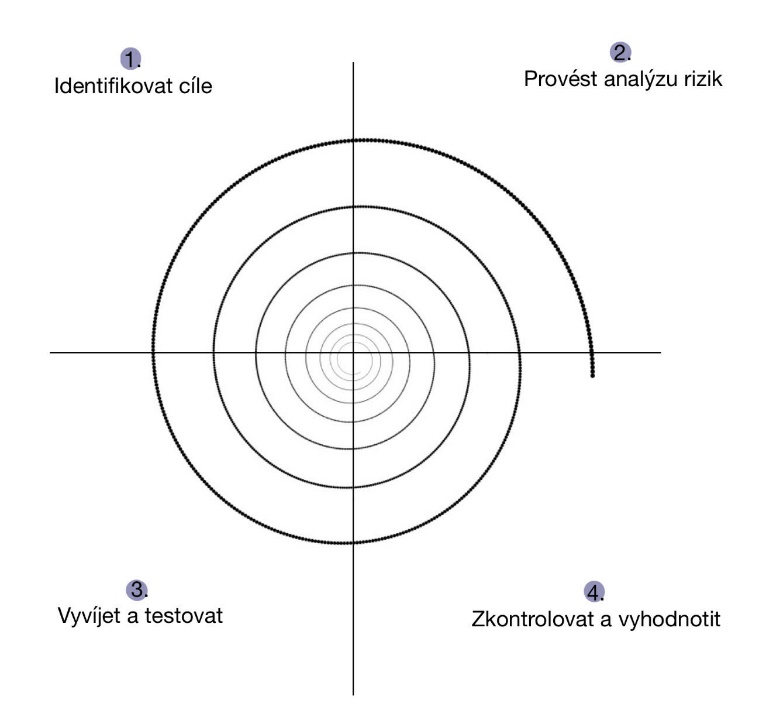
Model, ve kterém se počáteční vývojové práce provádějí na základě dobře stanovených základních požadavků a k tomuto základnímu software se prostřednictvím iterací přidávají následná vylepšení, dokud není vytvořen finální systém [6]. Vizualizace procesu je na obrázku 6.



Obrázek 6 Iterativní model

**Spirálový model**

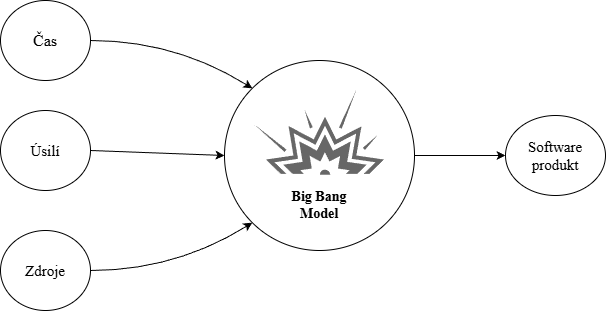
Spirálový model vyobrazen na obrázku 7 představuje kombinaci iterativního modelu a vodopádového modelu. Používá se především softwarovými inženýry a je preferován pro velké, drahé a složité projekty. Projektový manažer určuje počet smyček, které se liší v závislosti na projektu. Každá smyčka spirály je fází v modelu procesu vývoje software [7].



Obrázek 7 Spirálový model

**Big Bang**

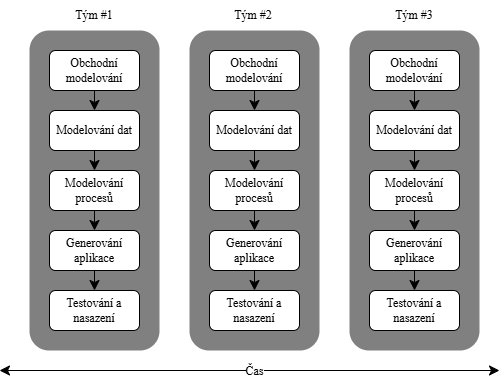
Big Bang je nejjednodušší model, protože nevyžaduje téměř žádné plánování. Vyžaduje však spoustu finančních prostředků a kódování a zabere více času. Tento model kombinuje čas, úsilí a zdroje pro vytvoření produktu. Produkt je postupně vytvářen podle požadavků zákazníka, nicméně konečný produkt nemusí splňovat skutečné požadavky [8]. Viz obrázek 8 znázorňující průběh tohoto modelu.



Obrázek 8 Big Bang model

**RAD (Rapid Application Development)**

Jak je patrné z obrázku 9 níže, RAD, neboli v překladu rychlý vývoj aplikací, je vývoj software založený na prototypování a rychlé zpětné vazbě s menším důrazem na specifické plánování. Obecně upřednostňuje vývoj a tvorbu prototypu před plánováním [9].



Obrázek Rad model

### Verzování vývoje

Verzování vývoje je proces sledování a řízení změn v softwarovém kódu. Systémy pro verzování vývoje jsou nástroje, které umožňují vývojovým týmům efektivně spravovat a uchovávat historii změn během celého vývojového procesu. Tyto nástroje usnadňují spolupráci více vývojářů na jednom projektu, umožňují sledovat jednotlivé úpravy   
a v případě potřeby se vrátit k předchozím verzím [10] [12].

Systémy pro správu verzí (VCS) se stále vylepšují a některé jsou lepší než jiné. Tyto systémy jsou někdy známe jako SCM (Source Code Management) nástroje nebo RCS (Revision Control Systém). Git je jedním z nejpopulárnějších VCS nástrojů, který se dnes používá. Git je bezplatný a open source. Ať už je použit jakýkoliv systém, hlavní výhody jsou následující [10]:

1. **Kompletní dlouhodobá historie změn souboru –** Vytváření, mazání a také úpravy jejich obsahu, to vše zahrnují změny. Historie by měl být také zaznamenán autor, datum a poznámky o účelu změny. Návrat k předchozím verzím v kompletní historii neschází, což pomáhá vyhledávat příčiny chyb,   
   a je potřebná při opravě problémů ve starších verzích software.
2. **Větvení a slučování –** Systém podporuje jak souběžnou práci týmů, tak i práci jednotlivce, který má možnost pracovat na nezávislých proudech změn. V VCS nástrojích lze vytvořit tak zvané „větve“ pro udržení více proudů práce na sobě nezávislých a zároveň poskytuje možnost sloučit práce zpět dohromady.
3. **Sledovanost** – S analýzou hlavních příčin a dalšími forenzními analýzami může pomoci schopnost sledovat každou změnu, která byla provedena na software a propojit ji se softwarem pro řízení projektů a sledování chyb, jako je Jira, a schopnost anotovat změny zprávou popisující účel a záměr provedené změny. Mít anotovanou historii může vývojářům umožnit provádět správné harmonické změny.

Verzovacích systémů je mnoho, ovšem zde je zaměřeno na systém Git. Je to nejrozšířenější moderní systém na světě. Má distribuovanou architekturu. Každý vývojář má zde pracovní kopii kódu, která může obsahovat historii všech změn projektu, což je výhodnější než v kdysi populárních systémech pro správu verzí, jako je CVS . Každou verzi projektu můžeme organizovat do tzv. větví, které můžou být jakkoliv pojmenovány. Je mnoho důvodů, proč požívat Git pro správu projektů [12]:

1. **Sledování změn** – Systém Git umožňuje ukládat veškeré změny a sledovat je. Tedy pokud něco v projektu není plně funkční nebo nefunguje správně a je třeba se vrátit k předchozí verzi, historie změn to umožňuje.
2. **Spolupráce** – Usnadňuje spolupráci vývojářů na jednom projektu. Každý člen týmu může pracovat na vlastní verzi a změny lze poté snadno sloučit.
3. **Větvení** – Vytvořením větví máte možnost pracovat nezávisle. To je užitečné pro opravu chyb a vývoj nových funkcí, aniž by se změny promítly do hlavního projektu.
4. **Zpětné vrácení změn** – Pokud se stane, že nové funkce obsahují problémy nebo provedete něco špatně, je možné se jednoduše vrátit k předchozímu stavu projektu.
5. **Ochrana dat** – Kdyby došlo k chybě nebo selhání disku, data jsou v bezpečí uložena v tomto systému.
6. **Podpora pro týmy** – Jelikož Git umožňuje sledování, kdo a kdy provedl změny, pomáhá tak efektivně koordinovat práci větších týmů.
7. **Open source a popularita –** Systém je open source, což znamená, že zdrojový kód je veřejně dostupný a bezplatný. Má širokou komunitu uživatelů a aktivní vývoj.
8. **Flexibilita** – Git není využíván pouze na vývoj software, ale také třeba i při psaní prací, správu obsahu webu a dalších.
9. **Rychlost** – Velikou výhodou je i rychlost a efektivita systému. Záznamy změn jsou úsporné, tudíž zabírají méně místa na úložišti.

Za základní stavební prvek principů Gitu je označován repozitář. Tedy místo, kde jsou uloženy a spravovány veškeré soubory, historie a metadata projektu. Navíc v něm můžou být uvedeny kompletní záznamy o změnách kódu, souborech, složkách   
a dalších součástí projektu [12].

Repozitář má hlavní funkci ukládat a udržovat celou historii projektu. Při uložení stavu projektu v určitém okamžiku vývoje provedeme tzv. commit. Commit je zaznamenán společně s autorem změny a kdy ke změně došlo v repozitáři. Což umožňuje sledovat vývoj projektu v čase, pohybovat se mezi verzemi a zjišťovat, kdo a kdy danou změnu provedl [12].

Důležitou součástí Gitu je jakým způsobem systém ukládá soubory. Každý soubor je uložen pouze jednou a poté se ukládají tzv. snapshoty. Veškeré soubory jsou v commitu uloženy jako snapshot [12].

Do Gitu je také možné ukládat i například obrázky či jiné netextové soubory díky tomu, že jsou uloženy všechny soubory binárně. Velikost repozitáře ovšem zůstává téměř identická [12].

Veškeré operace nejprve probíhají lokálně. Může se to zdát jako nevýhoda, protože se přidává krok mezi serverem a uživatele, ve skutečnosti to ale má spoustu výhod. Před tím, než se kód publikuje je snadné upravovat chyby na lokální práci. Jakmile se kód publikuje na server a někdo jiný ho stáhne, už není možné ho jakkoliv zpětně měnit [12].

Jelikož lze kód ukládat lokálně, není potřebný přístup k internetu. Téměř všechny operace lze provádět lokálně a data přidat na sever později, když je k dispozici síťové připojení [12].

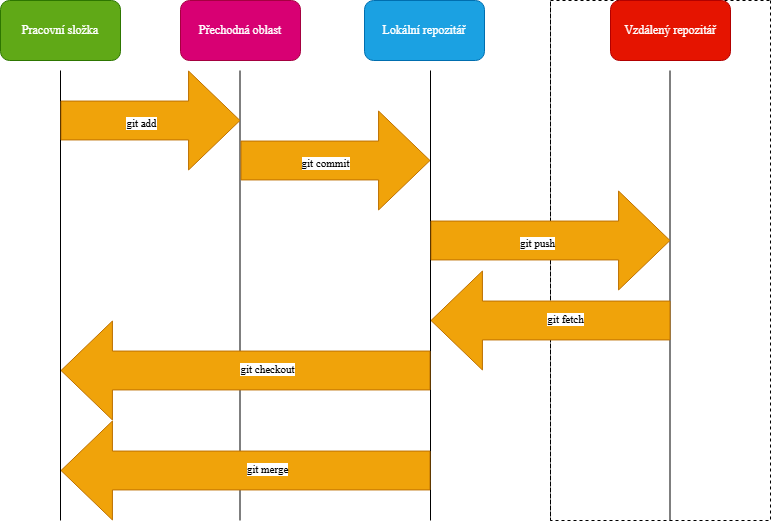
Git také klade velký důraz na integritu dat. Jakákoliv změna či poškození souboru je okamžitě detekováno. Systém ukládá soubory podle kontrolního součtu SHA-1, nikoli podle jména. Tento kontrolní součet provede sám Git pro každý soubor nebo složku [12].

Git funguje na principu přidávání dat. Tedy pokud bude odstraněn řádek, Git zaznamená informaci o odstranění a původní data zůstanou zachována. Je velmi obtížné poškodit data po provedení commitu natolik, aby už nebyla obnovitelná. Pokud nedojde k externí události, jsou soubory v repozitáři bezpečně uloženy [12].

Také je nutné zmínit, že celý proces správy verzí se skládá ze čtyř fází. Pomocí těchto fází se zvyšuje efektivita a koordinovanost spolupráce při verzování projektů. Jedná se o tyto čtyři fáze [12]:

1. **Pracovní složka** – Jedná se o místo, kde provádíme změny a editujeme projekt. Lze zde vytvářet, mazat a upravovat soubory dle svých potřeb.
2. **Přechodná oblast –** V tomto prostoru vybíráme, které změny budou obsahovat následující commit.
3. **Lokální repozitář** – Po vybrání souboru ve staging area provedeme commit, který uloží změny do lokálního repozitáře. V tomto repozitáři se vyskytují veškeré změny a commity, které byly provedeny.
4. **Vzdálený repozitář** – Pokud chceme, aby změny, které byly uloženy na lokální repozitář, viděli i ostatní spolupracovníci, nahrajeme tyto změny na vzdálený server.

Následující graf na obrázku 10 zobrazuje tento proces:



Obrázek 11 Proces Gitu

Na trhu je mnoho platforem systému Git. Mezi nejpoužívanější patří [12]:

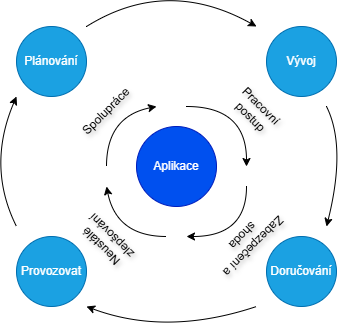
1. GitHub,
2. Bitbucket,
3. GitLab,
4. Azure DevOps,
5. SourceForge,
6. Beanstalk,
7. FramaGit.

### Proces DevOps

DevOps představuje propojení lidí, procesů a technologií s cílem zajistit kontinuální dodávání kvalitních produktů a služeb zákazníkům. Název vznikl spojením slov vývoj (development, Dev) a provoz (operations, Ops). Vývojáři, správci IT, testeři   
a specialisté na bezpečnost – dříve oddělené role a dnes pomocí DevOps spolu úzce spolupracují a koordinují své činnosti. Zavedením této kultury je umožněna rychlejší reakce na potřeby zákazníků, spolehlivost aplikací a přispívá k efektivnějšímu dosažení obchodních cílů organizace [13].

Životní cyklus aplikací je ovlivněn DevOps prostřednictvím jednotlivých fází, které jsou vyobrazené na obrázku 11. Každá fáze se spoléhá na ostatní, tedy každá role je v určité míře zapojená ve všech fázích. Těmito fázemi jsou [13]:

1. **Plánování** – V této fázi týmy vymýšlejí, definují a popisují funkce a možnosti projektů, které vytvářejí. Vytvářejí backlogy, sledují chyby a vizualizují průběh práce prostřednictvím řídících panelů.
2. **Vývoj** – Fáze vývoje zahrnuje veškeré kódování – od samotného psaní kódu, testování, kontrolu a integraci mezi členy týmu, až po tvorbu build artefaktů.
3. **Doručování** – Doručování je proces spolehlivého a konzistentního nasazování aplikací do produkčního prostředí. Fáze doručování také zahrnuje nasazení   
   a konfiguraci plně spravované infrastruktury, která zajišťuje funkčnost jednotlivých prostředí. Během této fáze týmy definují postup pro správu verzí s jasně definovanými postupy ručního schvalování. Také nastavují automatizované brány, které přesouvají aplikace mezi jednotlivými fázemi, před tím, než se zpřístupní zákazníkům.
4. **Provoz** – V provozní fázi se vývojáři zaměřují na údržbu a monitorování aplikací a řešení problémů s aplikací, která je již v provozu.



Obrázek 12 Proces DevOps

S DevOps úzce souvisí proces CI/CD znázorněný na obrázku 12 (Continuous Integration/Continuous Deployment), tedy kontinuální integrace a kontinuální dodávání či nasazení, představuje přístup, jehož cílem je urychlit a zefektivnit proces vývoje software [14].

Kontinuální integrace (CI) označuje proces automatické a časté integrace změn kódu do společného repozitáře, kde jsou následně testovány [14].

Kontinuální dodávání a nasazení (CD) navazuje na předchozí proces – jde o proces, který zahrnuje automatické testování, přípravu a vydávání nových verzí software [14].



Obrázek 13 Proces CI/CD

Tyto postupy se dohromady označují jako „kanál CI/CD“ a podporují je vývojové   
a provozní týmy, které spolupracují agilním způsobem s přístupem DevOps nebo SRE (site reliability engineering) [14].

Také je důležité zmínit, proč je CI/CD důležité. CI/CD pomáhá organizacím vyvarovat se chybám a selhání kódu a zároveň zajišťuje plynulý a nepřetržitý proces vývoje   
i aktualizace software [14].

Funkce CI/CD mohou pomoci snížit složitost aplikací, zvýšit efektivitu a zefetkivnit pracovní postupy [14].

Dříve tyto automatizované procesy byly prováděny manuálně při přenosu nového kódu z commitu do produkčního prostředí, čímž dochází k minimalizaci prostojů   
a urychlení vydávání aktualizací. Tento přístup také umožňuje častější a efektivnější začleňování zpětné vazby od zákazníků, což se pozitivně promítá do výsledků pro koncové uživatele a přispívá k větší spokojenosti zákazníků [14].

Vzhledem k ochraně kódových kanálů se používá zabezpečení CI/CD. Provádějí se automatizované kontroly a testování, aby se zabránilo zranitelnostem při dodávání software. Zabezpečení se dá začlenit do vašeho kanálu pomocí metod, jako je zabezpečení Shift Left a Shoft Right. Tyto metody pomáhají chránit kód před útoky, předcházet únikům dat, dodržovat zásady a zajišťovat kvalitu [14].

Pokud by se kanál nasazoval rychle a bez řádného zabezpečení může se vystavit několika rizikům, jako jsou [14]:

1. Zpřístupnění citlivých dat externím zdrojům.
2. Použití nezabezpečeného kódu nebo komponent třetích stran.
3. Neoprávněný přístup k úložištím zdrojového kódu nebo nástrojům pro sestavení.

Identifikace a zmírňování zranitelností v celém cyklu vývoje software zajišťuje, že změny jsou řádně testovány a splňují bezpečnostní standardy před nasazením do produkčního prostředí [14].

Je řada CI/CD nástrojů. Některé z nich se zaměřují především na část integrace (CI), jiné se soustředí naopak na vývoj a nasazování (CD) a další se specializují na kontinuální testování nebo související činnost [14].

Jeden z mnoha nástrojů je Tekton Pipelines, což je CI/CD framework určený pro platformy Kubernetes, který poskytuje standardní cloud-native prostředí pro CI/CD založené na kontejnerech. Kromě tohoto nástroje ovšem existuje několik dalších open source nástrojů, které stojí za pozornost [14]:

1. **Jenkins** – byl vytvořen tak, aby zvládl vše potřebné od jednoduchého CI serveru až po komplexní CD centrum.
2. **Spinnaker** – platforma určená pro multicloudová prostředí.
3. **GoCD** – CI/CD server s důrazem na modelování a vizualizaci procesů.
4. **Concourse** – „open-source nástroj pro nepřetržité provádění úloh“.
5. **Screwdriver** – platforma pro sestavování určená pro kontinuální dodávání.

Pro týmy mohou být také dobrou volbou spravované nástroje, které nabízejí různí poskytovatelé. Mezi hlavní veřejné cloudové služby patří například GitLab, CircleCI, Travis CI, Atlassian Bamboo a mnoho dalších [14].

Kromě toho je součástí CI/CD procesů často i několik běžně používaných nástrojů v DevOps. Nástroje pro automatizaci konfigurace (například Ansible, Chef, Puppet), běhová prostředí kontejnerů (například Docker, rkt, cri-o) a orchestrace kontejnerů (Kubernetes) sice nejsou přímo CI/CD nástroje, ale v těchto pracovních postupech jsou často využívané [14].

### Bezpečný vývoj

Pro každá organizaci, která usiluje o dodávání kvalitních produktů a aplikací je bezpečný vývoj software velmi zásadní. Vytváření bezpečných vývojových postupů je v týmech čím dál důležitější než kdy dříve vzhledem k rostoucímu počtu útoků. Bezpečnost se často jeví jako překážka, která brzdí vývoj, což je především kvůli tlaku na rychlé dodávání produktů a aplikací. Proto je mnohdy odkládána až na dobu po uvedení produktu na trh. Ovšem bezpečnost by měla být prioritou, a proto je nutné přijmout přístup security-by-design – tedy zaměřit se na bezpečnost již od začátku vývoje [15].

Základem bezpečného vývoje je životní cyklus bezpečného vývoje software (Secure SDLC), což je sled fází, kterými software prochází během vývoje [15].

Pomocí každé fáze Secure SDLC mohou organizace výrazně snížit bezpečnostní rizika spojená se softwarem, který uvádějí na trh. Pokud týmy zavedou princip security-by-design již v raných fázích předejdou nutnosti přepracování a zpětným úpravám. Tím se stává řešení bezpečnostních problémů výrazně levnějším [15].

Proces bezpečného vývoje integruje bezpečnost do všech fází životního cyklu vývoje. Proces začíná plánováním, kde se stanovují bezpečnostní požadavky, zohledňuje se soulad s předpisy a obchodní cíle, přiděluje se rozpočet a definují se relevantní standardy a rámce [15] [16].

Následuje fáze požadavků, ve které se shromažďují požadavky na nové funkce od různých zainteresovaných stran. Je důležité už v tomto kroku identifikovat případné bezpečnostní aspekty, které se týkají funkčních požadavků připravovaného vydání software[16] [17].

Ve fázi návrhu se předkládají definované požadavky do konkrétního plánu, jak by měla aplikace vypadat a fungovat v praxi. Funkční požadavky obvykle popisují, co by se mělo stát, zatímco bezpečnostní požadavky se naopak zaměřují na to, co by se stát nemělo [16] [17].

V této fázi je systematický proces modelování hrozeb. Slouží k identifikaci, komunikaci a pochopení hrozeb a způsobu jejich zmírnění v kontextu ochrany něčeho hodnotného. Model hrozeb představuje strukturované zobrazení všech informací, které ovlivňují bezpečnost aplikace. V podstatě jde o pohled na aplikaci a její prostředí z hlediska bezpečnosti. Modelování hrozeb lze použít na širokou škálu oblastí   
– například na software, aplikace, systémy, sítě, distribuované systémy, zařízení internetu věcí nebo na obchodní procesy [16] [18].

Následuje fáze implementace, kdy vývojáři software píší bezpečný kód s využitím bezpečnostních požadavků z plánovací fáze a bezpečnostních kontrol   
a architektonických prvků přidaných ve fázi návrhu. Aby se vývojáři vyhnuli vzniku bezpečnostních problémů, používají osvědčené postupy. Statická analýza kódu (SAST) a analýza softwarového složení (SCA) pomáhají identifikovat a řešit bezpečnostní nedostatkyjiž během implemetnace [15] [16].

Analýza složení software (SCA) je automatizovaný proces, který identifikuje open-source software v dané kódové základně. Tato analýza se provádí za účelem vyhodnocení bezpečnosti, souladu s licencemi a kvality kódu [16] [19].

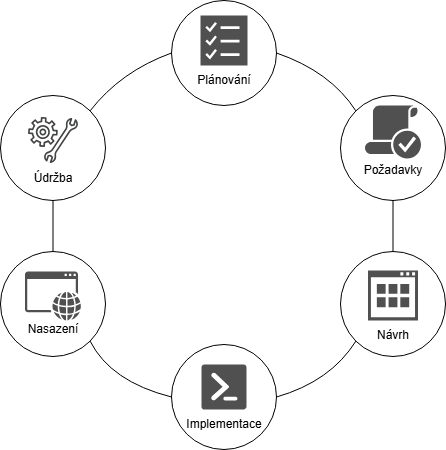
S fází implementace také úzce souvisí Software Bill of Materials (SBOM). Jedná se   
o seznam všech komponent, knihoven a modulů v softwarovém produktu ve strojově čitelném formátu. Pomáhá organizacím sledovat jednotlivé prvky software, identifikovat zranitelnosti a snižovat bezpečnostní rizika v rámci celého dodavatelského řetězce [16] [20].

Je třeba také zmínit kryptografii. Kryptografie je základem digitální bezpečnosti   
a ochrany soukromí. V době, kdy se organizace snaží modernizovat svou bezpečnostní infrastrukturu, je klíčové porozumět tomu, jaké kryptografické komponenty se v jejich systémech nacházejí a jak jsou nakonfigurovány. V této fázi přichází CBOM, který vychází ze Software Bill of Materials, ale zaměřuje se konkrétně na kryptografické prvky integrované do softwarových aplikací [16] [21].

Další fází je fáze nasazení. Nasazení je moment, kdy je software uveden do provozu. V této fázi je důraz kladen hlavně na to, aby byl celý proces nasazení bezpečný a aby byla přijata opatření zabraňující případným bezpečnostním problémům během tohoto kroku [16] [22].

Celý cyklus uzavírá fáze údržby, během které týmy pravidelně kontrolují a aktualizují bezpečnostní opatření, aby minimalizovaly nové a vznikající bezpečnostní hrozby. K řešení některých problémů mohou být nutné bezpečnostní záplaty. Software je neustále monitorován, což umožňuje týmům rychle reagovat na případné bezpečnostní incidenty [15] [16].

Následující obrázek 13 zobrazuje proces SSDLC.



Obrázek 14 Proces SSDLC

## Softwarové zranitelnosti

Softwarová zranitelnost je strukturní nebo návrhová chyba v aplikaci, kterou mohou útočníci zneužít k ohrožení bezpečnosti a funkčnosti systému, sítě nebo dat, se kterými aplikace interaguje. Zranitelnosti vznikají pomocí chybám v kódu, přehlédnutím v návrhu, zastaralému software či neúmyslným interakcím mezi komponentami   
a mohou mít dalekosáhlé důsledky pro bezpečnost a stabilitu digitálního ekosystému.

Pravděpodobnost a dopad softwarových zranitelností lze snížit několika způsoby [23]:

1. **Testování zranitelností a správa záplat** – Je nezbytné pravidelně hodnotit zranitelnosti a provádět penetrační testy. K tomu může pomoci řada automatizovaných nástrojů.
2. **Hodnocení rizik a školení v oblasti bezpečnosti** – Je důležité vytvořit prostředí, kde si každý uvědomuje bezpečnostní rizika. To znamená kombinovat hodnocení rizik a školení zaměstnanců. Školení by mělo být praktické   
   a komplexní, seznamovat týmy s běžnými hrozbami a učit je, jak se jim bránit. Pravidelné hodnocení rizik pak umožňuje sledovat zranitelnosti a odhadovat, jaký dopad by mohly mít na podnikání.
3. **Nepřetržité monitorování a pravidelné zálohování** – Pro efektivní řešení zranitelností je nutné mít plán monitorování a zálohování. Pokud tento plán tým má, může díky tomu produkt pokračovat v provozu i při kritických bezpečnostních incidentech. Při implementaci kontinuálního monitorování všech softwarových systémů je jednoduché odhalovat škodlivé nebo podezřelé aktivity. Dále je nutné provádět pravidelné zálohování dat a systémů. Zálohy umožňují rychlé obnovení po zneužití zranitelností.

### Známe zranitelnosti (CVE)

Veřejně známe zranitelnosti neboli CVE (Common Vulnerabilities and Exposures), jsou katalogem zranitelností informační bezpečnosti, který je veřejně dostupný   
a spravovaný organizací MITRE Corporation [26].

Katalog CVE je spíše označován jako slovník než databáze. Vyskytují se zde jména   
a popisy jednotlivých zranitelností nebo expozic. Čímž umožňuje komunikaci mezi různými nástroji a databázemi a pomáhá zvýšit bezpečnost. CVE seznam lze stáhnout zdarma a použít [26].

Také je nutné zmínit, že zranitelnost a expozice není to stejné. Zranitelnost je chyba v programu nebo systému, kterou může útočník zneužít k neoprávněnému přístupu. Pomocí ní mohou spouštět škodlivý kód, dostat se do paměti systému, instalovat malware nebo krást, mazat či měnit citlivá data. Expozice je také chyba v programu nebo nastavení, ale útočníkovi už umožňuje přímý přístup do systému nebo sítě. Může způsobit únik dat, poručení bezpečnosti, a dokonce prodej osobních údajů na dark webu [25].

Jelikož jedním ze základních problémů v kybernetické bezpečnosti je identifikace   
a zmírnění zranitelností CVE pomáhá tento problém efektivně řešit. CVE poskytuje standardizovaný rámec pro kategorizaci a sledování kybernetických zranitelností, který organizace mohou využít ke zlepšení procesů správy zranitelností [26].

Systém CVE využívá jedinečné identifikátory, známé jako CVE ID, k označení každé nahlášené zranitelnosti. Tyto identifikátory jsou přiřazovány CVE Numbering Authority (CNA) – orgány, které mají oprávnění přidělovat CVE. Existuje přibližně 100 těchto orgánů, mezi nimiž jsou bezpečnostní společnosti, výzkumné organizace   
a IT dodavatelé, jako jsou Red Hat, IBM, Cisco, Oracle a Microsoft [24].

Každý záznam CVE obsahuje CVE ID, stručný popis zranitelnosti a odkazy na zprávy o zranitelnostech nebo doporučení. CVE ID mají tříčlennou strukturu: začínají předponou „CVE“, následuje rok přiřazení a na konci je postupné číslo [26].

Záznamy CVE mohou být ve třech stavech. Stav Reserved (Rezervováno) znamená, že CVE bylo přiřazeno, ale ještě nebylo zveřejněno. Stav Published (Publikováno) nastává, když CNA shromáždí potřebná data a záznam je zveřejněn. Stav Rejected (Odmítnuto) označuje, že CVE ID a záznam by neměly být používány, ovšem záznam stále zůstává v záznamu, aby uživatelé věděli, že je neplatný [26].

Příkladem může být CVE-2023-23397. Jedná se o zranitelnosti Microsoft Outlook umožňující zaslání speciálně upraveného e-mailu, který se spustí automaticky při jeho stažení a zpracování Outlook klientem.[27] Nebo zranitelnost CVE-2025-55177, která pojednává o neúplné autorizaci zpráv synchronizace propojených zařízení v aplikaci WhatsApp pro iOS mohla umožnit nesouvisejícímu uživateli spustit zpracování obsahu z libovolné adresy URL na cílovém zařízení [28].

Jednou z nejznámějších zranitelností je CVE-2017-0144. Umožňuje vzdáleným útočníkům spouštět libovolný kód. Známý je především proto, že tato zranitelnost byla zneužita skupinou kyberzločinců v květnu roku 2017. Tento červ se rozšířil na více než 200 000 počítačů ve více než 150 zemích. Mezi známé oběti patřily FedEx, Honda, Nissan a britská Národní zdravotní služby, která byla nucena přesměrovat některé své sanitky do jiných nemocnic [29] [30].

### Kategorizace zranitelností (CWE)

Je systém klasifikace a kategorizace běžných typů zranitelností. V současnosti existuje více než 600 kategorií, sahajících od přetečení bufferu, cross-site scripting až po nezabezpečená náhodná čísla [31].

Slabiny jsou většinou zranitelnosti, které mohou zahrnovat chyby, bugy nebo jiné nedostatky v hardware či software, v kódu, návrhu či architektuře. Tyto zranitelnosti vytvářejí potenciální expozici vůči kybernetickému útoku. Tento seznam je organizován taxonomií, která usnadňuje nalezení, identifikaci a popis slabin způsobem, kterému rozumí celá komunita [31].

Cílem CWE je eliminovat zranitelnosti tak, že identifikuje nejčastější chyby, které vývojáři a inženýři dělají, aby se jim mohli vyhnout při tvorbě produktů a systémů. CWE poskytuje přehlednou taxonomii a jednotný jazyk, pomáhá vývojářům kontrolovat slabiny v existujícím software a produktech a nabízí další nástroje pro prevenci zranitelností [31].

CWE je komunitně vyvíjený projekt, do kterého se zapojují účastníci z průmyslu   
i vlády. Program CWE je sponzorován americkým Ministerstvem vnitřní bezpečnosti (DHS) a Cybersecurity and Infrastructure Security Agency (CISA) a spravován Homeland Security Systems Engineering and Development Institute (HSSEDI), který provozuje MITRE Corporation [31].

Department of Homeland Security (DHS) má za úkol chránit USA před různými hrozbami.

CISA spolupracuje s partnery na obranně proti hrozbám a budování bezpečnější   
a odolnější infrastruktury. Například vede ochranu federální domény „.gov“ pro civilní vládní sítě a spolupracuje s privátním sektorem na zvyšování bezpečnosti kritických sítí [31].

Jednou z nejvíce nebezpečných slabin je slabina CWE-79. Jedná se o Cross-Site Scripting (XSS), což je útok typu injekčního útoku, při kterém jsou škodlivé skripty vloženy do jinak důvěryhodných webových stránek. K útokům XSS dochází, když útočník využije webovou aplikaci k odeslání škodlivého kódu, obvykle ve formě skriptu běžícího na straně prohlížeče, jinému koncovému uživateli [32][33]. Následující obrázek 14 představuje proces XSS útoku.



Obrázek 15 – Průběh XSS útoku

### Hodnocení rizik (CVSS)

Jak již bylo zmíněno CVE je seznam nebo slovník veřejně známých bezpečnostních chyb. Tyto chyby jsou hodnoceny a skórovány pomocí Common Vulnerability Scoring Systém (CVSS) [34].

CVSS je standardizovaný rámec pro měření závažnosti bezpečnostních chyb v informačních systémech. Každé zranitelnosti přiřazuje skóre od 0 do 10, přičemž vyšší skóre znamená závažnější problém. Tento systém pomáhá organizacím rozhodnout, které bezpečnostní hrozby je třeba řešit nejdříve na základě jejich potenciálního dopadu [34].

Skóre je přiřazováno na základě tří hlavních faktorů: Base, Temporal a Environmental. Base ukazuje základní vlastnosti zranitelnosti, Temporal zohledňuje, jak se tyto vlastnosti mohou v průběhu času měnit a Environmental hodnotí, jak by zranitelnost mohla ovlivnit konkrétní prostředí [34].

Tabulka 1 znázorňuje rozdělení jednotlivých úrovní závažnosti podle CVSS verze 4.0 [34]:

Tabulka 1 Úrovně závažnosti podle CVSS

|  |  |
| --- | --- |
| ***Skóre*** | ***Hodnocení na základě závažnosti*** |
| 0,0 | Žádná |
| 0,1–3,9 | Nízká |
| 4,0–6,9 | Střední |
| 7,0–8,9 | Vysoká |
| 9,0–10,0 | Kritická |

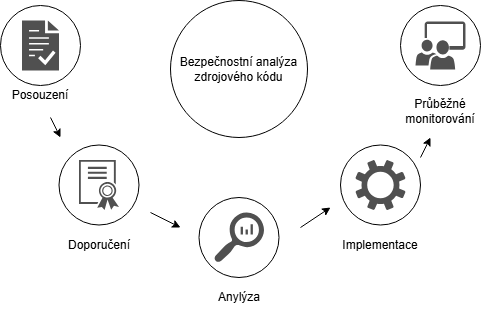
## Bezpečnostní analýza zdrojového kódu

Bezpečnostní analýza kódu je ruční nebo automatizovaný proces, při kterém se analyzuje zdrojový kód aplikace s cílem odhalit bezpečnostní chyby či zranitelnosti. Kontrola se zaměřuje například na logické chyby, správnou implementaci specifikací nebo dodržování zásad psaní kódu [35].

Automatizovaná kontrola kódu využívá nástroje, které provádějí revizi zdrojového kódu podle předem definovaných pravidel. Tento přístup umožňuje rychlejší identifikaci potenciálních problémů než manuální kontrola [35].

Manuální kontrola kódu spočívá v pečlivém procházení zdrojového kódu člověkem, řádek po řádku, s cílem odhalit bezpečnostní slabiny. Manuální revize dokáže zohlednit kontext rozhodnutí vývojáře a obecnou logiku aplikace, což automatizované nástroje často nedokážou. Tento přístup je strategičtější a umožňuje řešit konkrétní problémy cíleně [35].

Proces bezpečné analýzy zdrojového kódu je znázorněn na obrázku 15.



Obrázek 16 Proces bezpečné analýzy zdrojového kódu

### Statická analýza

Statická analýza kódu se obvykle provádí jako součást kontroly kódu, známé také jako white-box testing neboli testování s přístupem ke zdrojovému kódu. Tato analýza probíhá ve fázi implementace v rámci životního cyklu vývoje zabezpečeného software [37].

Statická analýza kódu většinou znamená spuštění nástrojů pro statickou analýzu. Nástroje se snaží odhalit možné zranitelnosti v neprobíhajícím zdrojovém kódu pomocí technik, jako jsou taint analýza a analýza toku dat [37].

Statická analýza se dělí na lexikální analýzu, syntaktickou analýzu, sémantickou analýzu, symbolickou exekuci, analýzu toku dat a graf toku řízení.

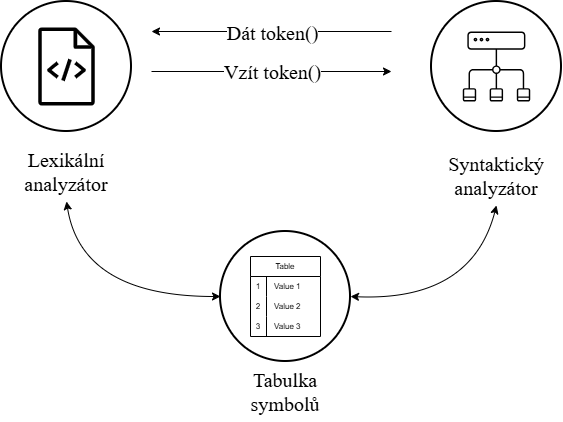
Lexikální analýza znázorněná na obrázku 16 je první fází kompilátoru, která spočívá v čtení zdrojového programu znak po znaku zleva doprava a jeho uspořádání do tokenů[[1]](#footnote-1). Lexikální analyzátor používá deterministický konečný automat (DFA) k rozpoznávání těchto tokenů, protože DFA jsou navrženy tak, aby rozpoznávaly regulární jazyky. Každý koncový stav odpovídá určitému typu tokenu, což umožňuje skeneru správně klasifikovat vstup. Proces vytváření DFA z regulárních výrazů lze automatizovat, což usnadňuje a zefektivňuje rozpoznávání tokenů. Lexikální analyzátor dokáže odhalit chybu pomocí automatového stroje a gramatiky daného jazyka. Při chybě dokáže určit řádek i sloupec, kde se nachází [39][42].

Příklad lexikální analýzy:

* Vstup: int exampleValue = 2 \* 7;
* Výstup znázorňuje tabulka 2 [42].

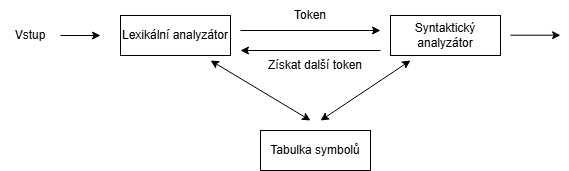
Tabulka 2 Výstup lexikální analýzy

|  |  |  |
| --- | --- | --- |
| **Token** | **Typ** | **Význam** |
| Int | Datový typ | Celé číslo |
| exampleValue  =  2  \*  7  ; | Identifikátor  Operátor  Literál  Operátor  Literál  Oddělovač | Název proměnné  Přiřazení  Celé číslo 2  Násobení  Celé číslo 7  Konec výrazu[[2]](#footnote-2) |



Obrázek 17 Proces lexikální analýzy

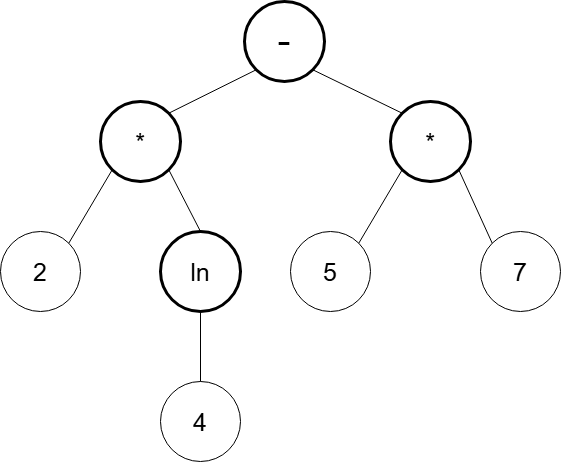
Syntaktická analýza neboli parsování či analýza syntaxe je fáze zpracování přirozeného jazyka. Cílem této fáze je odvodit přený význam textu, nebo lze říci „slovníkový význam“. Syntaktická analýza kontroluje, zda text dává smysl podle pravidel formální gramatiky [38]. Obrázek Obrázek 17 ukazuje průběh syntaktické analýzy.



Obrázek 18 Proces syntaktické analýzy

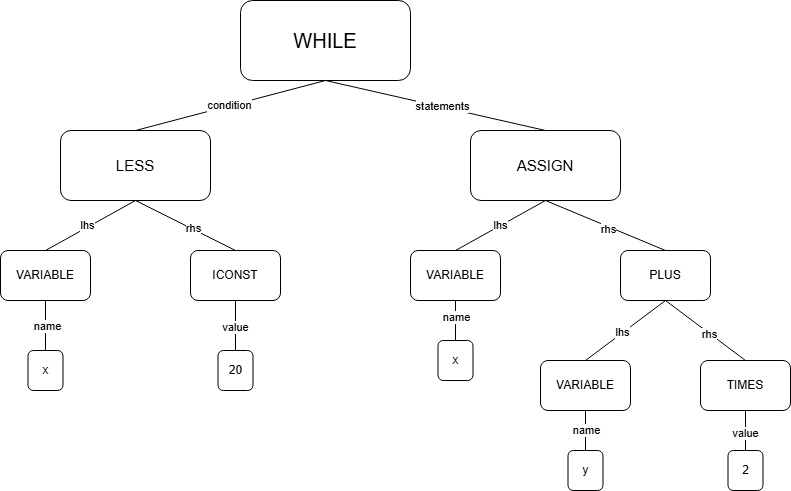
Příklad abstraktního syntaktického stromu je vyobrazen na obrázku 18 [42].

Výraz: 2 \* ln(4) – 5 \* 7



Obrázek 19 Příklad abstraktního syntaktického stromu

Na obrázku 19 lze vidět příklad vizualizace abstraktního syntaktického stromu [42].



Obrázek 20 Vizualizace abstraktního syntaktického stromu

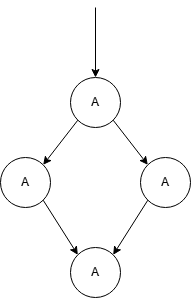
Sémantická analýza je podoblast zpracování přirozeného jazyka, která se snaží porozumět významu přirozeného jazyka. Porozumění přirozenému jazyku se člověku může zdát jako samozřejmost a jednoduchý proces. Každopádně kvůli obrovské složitosti a subjektivitě, která je s lidským jazykem spojena, je jeho interpretace pro stroje poměrně náročný úkol. Sémantická analýza zachycuje význam daného textu s ohledem na kontext, logickou strukturu vět a gramatické vztahy [41][42].

Fáze symbolické exekuce je fáze, která systematicky analyzuje všechny možné cesty programu, což vývojářům umožňuje odhalit chyby, bezpečnostní nedostatky. Nahrazením konkrétních hodnot symbolickými proměnnými dokáže symbolická exekuce zkoumat více scénářů vykonávání současně, čímž zajišťuje větší pokrytí kódu. Velmi užitečné je to při automatickém generování testů, detekci zranitelností   
a ověřování software. Symbolická exekuce má spoustu výhod, ale také nevýhod jako třeba exploze počtu cest, složité řešení omezujících podmínek a problémy s škálovatelností [42][63].

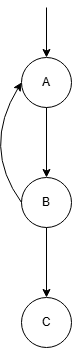
Technika analýza toku dat se používá při návrhu překladačů k analýze toho, jak data procházejí programem. Sleduje hodnoty proměnných a výrazů během jejich výpočtu a používá se v programu s cílem identifikovat možnosti optimalizace a odhalit potenciální chyby. Analýza toků dat především modeluje graf, kde uzly představují příkazy programu a hrany znázorňují závislosti toku dat mezi jednotlivými příkazy. Informace o toku dat jsou následně šířeny tímto grafem pomocí sady pravidel a rovnic, které slouží k výpočtu hodnot proměnných a výrazů v každém bodě analyzovaného programu [42] [64].

Graf toku řízení vizuálně znázorňuje tok řízení programu neboli to, jak se program pohybuje mezi různými příkazy a rozhodovacími body. V tomto grafu každý uzel představuje základní blok – souvislou část kódu, která má jen jeden vstup a jeden výstup a uvnitř ní se nevyskytují žádné skoky ani větvení. Základní bloky začínají tam, kam mohou směřovat skoky, a končí instrukcemi, které provádějí skok nebo větvení [65].

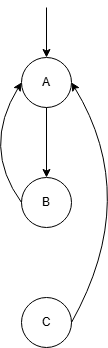
Graf toku řízení může být například podmíněné větvení vyobrazené na obrázku 20, na obrázku 21 je cyklus znázorňující iterativní strukturu (např. cyklus while) a na obrázku 22 je též cyklus [42].



Obrázek 21 Graf toku řízení – podmíněné větvení



Obrázek 22 Graf toku řízení – cyklus



Obrázek 23 Graf toku řízení – cyklus se zpětnou hranou do A

### SAST

SAST neboli statické testování bezpečnosti aplikací je bezpečnostní technika určená k analýze zdrojového kódu, bajtkódu či binárních souborů aplikace za účelem odhalení zranitelností, aniž by bylo nutné program spouštět. Díky tomu, že funguje v kontextu před spuštěním programu, mohou SAST nástroje identifikovat problémy jako nebezpečné volání API, hardcodovaná hesla nebo nebezpečné způsoby zpracování dat, než se stanou aktivními hrozbami [38].

Cílem SAST je umožnit vývojovým týmům řešit zranitelnosti již v jejich počátcích. Díky včasné identifikaci problémů v rámci životního cyklu vývoje software (SDLC) SAST snižuje náklady a složitost jejich odstranění. Tato analýza pomáhá zabránit šíření bezpečnostních chyb do dalších fází vývoje, čímž se snižuje riziko jejich zneužití v produkčním prostředí. Integrace SAST do CI/CD pipeline navíc zajišťuje průběžnou zpětnou vazbu pro vývojáře, podporuje kulturu bezpečného kódování a umožňuje týmům udržovat vysoké bezpečnostní standardy při vývoji software [38].

SAST nástroje nejprve analyzují kódovou základnu a vytvářejí abstraktní syntaktický strom (AST) nebo graf toku řízení (CFG), které modelují strukturu a chování aplikace. Tyto reprezentace umožňují nástroji provádět následující kroky [38]:

1. **Lexikální analýza** – Analyzuje zdrojový kód řádek po řádku a identifikuje syntaktické vzory odpovídající předdefinovaným bezpečnostním pravidlům. Například může upozornit na nebezpečné použití funkcí jako je eval() nebo odhalit hardcodovaná hesla.
2. **Sémantická analýza** – Tato analýza rozumí vztahům a tokům dat v kódu. Tento krok pomáhá odhalit zranitelnosti, jako je SQL injection, kdy neregistrovaný vstup od uživatele prochází do databázového dotazu bez patřičné validace.
3. **Analýza toku dat** – Sleduje, jak se data pohybují v aplikaci, se zaměřením na zdroje (např. vstupy od uživatele), cíle (např. dotazy do databáze nebo systémové volání) a transformace (např. funkce pro sanitizaci). Nástroj například dokáže zjistit, zda vstup od uživatele vstupuje do SQL dotazu bez sanitizace.
4. **Analýza znečištění** – Označuje nedůvěryhodné vstupy jako „znečištěné“   
   a sleduje jejich šíření kódem. Tím se odhalují situace, kdy nedůvěryhodná data ovlivňují citlivé operace, jako je přístup k souborovému systému nebo spouštění příkazů.
5. **Porovnání vzorů a aplikace pravidel** – Používá knihovnu bezpečnostních pravidel k identifikaci částí kódu, které porušují osvědčené postupy. Pravidla mohou být obecná nebo specifická pro jazyk.
6. **Analýza toku řízení a závislosti** – Zkoumá spouštěcí cesty a externí závislosti aplikace. Tím lze odhalit zranitelnosti způsobené nebezpečnými knihovnami třetích stran nebo nechráněnými podmíněnými větami.

### Srovnání vybraných nástrojů

Následující tabulka 3 vyobrazuje porovnání vybraných open-source SAST nástrojů podle zvolených kritérií.

Tabulka 3 Srovnání SAST nástrojů

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **SAST nástroj** | **Jazyky** | **GUI** | **CI/CD** | **Reporty** | **Cloud** | **IDE** |
| CodeQL [44][45][46] | 10 | Ano[[3]](#footnote-3) | Ano[[4]](#footnote-4) | Ano | Ne | Ano |
| GitLab SAST [47][48] | 19 | Ano[[5]](#footnote-5) | Ano | Ano | Ano | Ano |
| SonarQube [49][50][51] | 40 | Ano | Ano | Ano | Ano | Ano |
| Snyk [52][53][54][55] | 14 | Ano | Ano | Ano | Ano | Ano |
| Bearer [56][57][58] | 10[[6]](#footnote-6) | Ne | Ano | Ano | Ano | Ne |
| Aikido [59][60] | 16 | Ano | Ano | Ne | Ano | Ano |
| Horusec [61][62] | 18 | Ano | Ano | Ano | Ne | Ano |

Z porovnání vyplývá, že každý z vybraných nástrojů podporuje alespoň 10 programovacích jazyků. Nejvyšší počet podporovaných programovacích jazyků nabízí SonarQube, a to celkem 40. Všechny analyzované nástroje, až na Bearer, poskytují grafické uživatelské rozhraní (GUI), tedy přehledné vizuální prostředí, které umožňuje práci s nástrojem bez nutnosti používat příkazovou řádku. Integrace do CI/CD je podporována u všech z uvedených nástrojů. Možnost generování reportů je dostupná ve všech případech kromě nástroje Aikido. Cloudové řešení nenabízí pouze Horusec   
a CodeQL a co se týče podpory integrovaného vývojového prostředí (IDE), chybí výhradně u nástroje Bearer.

# Praktická část

Následující kapitola se věnuje praktickému nasazení nástrojů pro statickou analýzu zdrojového kódu (SAST). V úvodu kapitoly jsou nejprve představeny použité nástroje, včetně jejich rozdělení na nástroje provozované v cloudovém prostředí a nástroje nasazované lokálně. Zároveň jsou zde uvedeny podpůrné technologie využité v rámci práce, konkrétně Docker, Windows Subsystem for Linux (WSL) a repozitáře na platformách GitHub a GitLab. Dále se práce zaměřuje na přípravu testovacího prostředí, ve které je popsán výběr zranitelné webové aplikace. Součástí této části je rovněž popis instalace a základní konfigurace nástrojů potřebných pro realizaci praktické části. Dále se praktická část zaměřuje na samostatnému testování jednotlivých SAST nástrojů. Každý nástroj je samostatně nasazen a spuštěn nad zvoleným zranitelným zdrojovým kódem, přičemž jsou zaznamenány a analyzovány jeho výstupy. Závěrem praktické části je vyhodnocení získaných výsledků a jejich vzájemné porovnání. Důraz je kladen na úspěšnost detekce zranitelností, míru falešně pozitivních nálezů a schopnost nástrojů identifikovat vybrané typy zranitelností. Na základě těchto výsledků je vedena diskuze a formulována doporučení pro efektivní využití SAST nástrojů v praxi.

## Výběr technologií

Pro testování a porovnání jsou použity nástroje, které byly již uvedeny v kapitole 1.3.3, jedná se konkrétně o CodeQL, GitLab SAST, SonarQube, Snyk, Bearer, Aikido a Horusec. Tyto nástroje jsou využity k analýze zranitelného kódu webové aplikace a následně porovnány na základě předem stanovených hodnoticích metrik. Z hlediska způsobu nasazení jsou nástroje rozděleny na cloudová a lokálně provozovaná řešení. Cloudově jsou využity nástroje GitLab SAST, Snyk, Aikido a Bearer, zatímco lokálně jsou nasazeny nástroje CodeQL, SonarQube a Horusec. Toto rozdělení umožňuje porovnat rozdíly mezi cloudovým a lokálním přístupem ke statické analýze zdrojového kódu. Pro zajištění jednotného, izolovaného a opakovatelného testovacího prostředí jsou v rámci praktické části využity technologie Docker a Windows Subsystem for Linux (WSL).

## Příprava prostředí

V rámci projektu je důležité vybrat zranitelnou webovou aplikaci. Statická analýza je provedena na velmi známé a široce používané webové aplikaci OWASP Juice Shop [66]. Tato aplikace je vyvíjena organizací OWASP (Open Web Application Security Project) a slouží především k výuce a testování webových bezpečnostních zranitelností, které odpovídají nejčastějším rizikům definovaným v seznamu OWASP Top 10.

Zvolená aplikace je forknuta do vlastního repozitáře na platformách GitHub a GitLab, aby bylo možné s projektem pracovat nezávisle na původním autorovi a bezpečně provádět úpravy bez zásahu do originálního repozitáře.

Pro vytvoření vhodného vývojového a testovacího prostředí je na operačním systému Windows nainstalována technologie Windows Subsystem for Linux (WSL). Jako linuxová distribuce je použita Ubuntu 24.04 LTS, která poskytuje stabilní a aktuální prostředí pro práci s linuxovými nástroji.

Současně je nainstalována i technologie Docker, která umožňuje spouštění aplikací v kontejnerech. Docker je využíván k nasazení zranitelností aplikace do izolovaného prostředí, což je z hlediska bezpečnostního testování klíčové.

## Testování SAST nástrojů

V následujících podkapitolách je popsán proces nasazení, spuštění a vyhodnocení vybraných SAST nástrojů.

### GitLab SAST

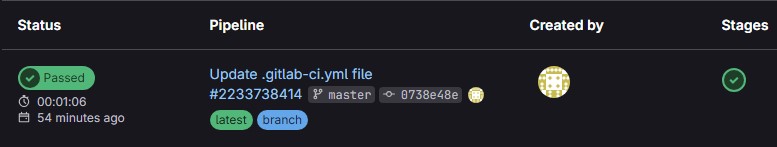
Prvním krokem k úspěšné analýze je nastavení automatizace v nástroji GitLab CI/CD. V kořenovém adresáři projektu je nutno vytvořit konfigurační *soubor .gitlab-ci.yml*. Tento soubor slouží jako instrukční sada pro GitLab, která mu říká, jaké úkoly má provádět. Do těchto instrukcí pro statickou analýzu spadá část *stages*, kde je vytvořen „šuplík“, v tomto projektu je pod názvem *test*, dále je část instrukcí *sast*, tedy vlastní název konkrétní práce, důležitou součástí je *stage: test*, která propojuje konkrétní práci s časovým plánem a říká GitLabu, že tato analýza má proběhnout ve fázi testování a jako poslední je *include*, kde je uvedena cesta k šabloně pro bezpečnostní analýzu. Tento kód je vyobrazen ve výpisu 1.

Výpis 1 Kód konfigurace statické analýzy v GitLab CI/CD



Po uložení změn GitLab automaticky spustí tzv. *Pipeline*. V rámci fáze test se aktivuje *job* neboli úkol pojmenovaný *semgrep-sast*. Tento skener pracuje na principu statické analýzy, což znamená, že nezkoumá běžící aplikaci, ale prochází zdrojový kód jako text. Hledá v něm specifické vzorce a programátorské chyby, které odpovídají známým bezpečnostním slabinám.

Průběh analýzy je možné sledovat v reálném čase skrze konzoli daného úkolu. Po dokončení všech testů se stav *Pipeline* změnil na zelené *Passed*, jako je vidět na obrázku 23, což potvrdí, že skenovací nástroje úspěšně prošly celý projekt a vygenerovaly finální zprávu (report). Celý proces, který zahrnuje přípravu prostředí, stažení kódu, samotnou analýzu a odeslání výsledků, proběhl za pouhou 1 minutu a 6 sekund. Tento čas je také vidět na obrázku Obrázek 23 u ikonky stopek.



Obrázek 24 Úspěšné provedení analýzy nástroje GitLab

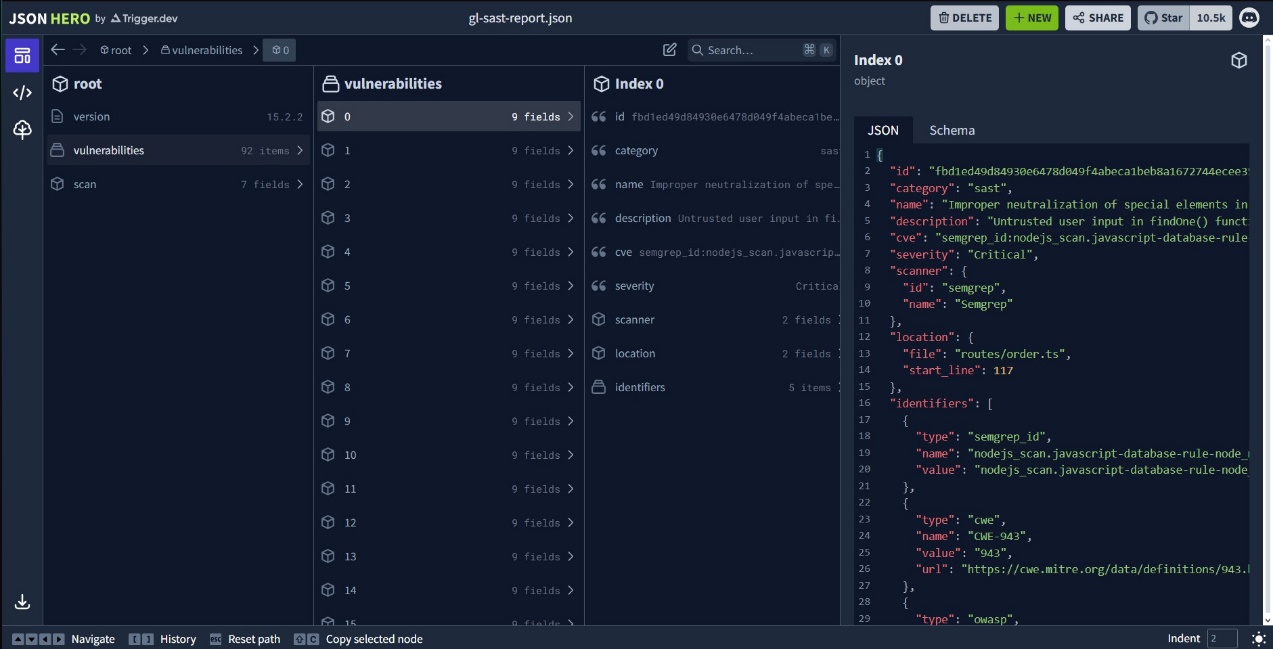
Jelikož bezplatná verze GitLabu standardně nenabízí grafické rozhraní pro zobrazení výsledků, bylo nutné výsledky získat přímo z výstupních dat. Skener vygeneroval soubor *gl-sast-report.json*, který byl uložen v artefaktech daného úkolu. Tento soubor představuje surový, strojově čitelný seznam všech nalezených bezpečnostních hrozeb. Část tohoto souboru lze vidět ve výpisu 2**Chyba! Nenalezen zdroj odkazů.**.

Výpis 2 Část výstupu analýzy GitLab



Z důvodu obtížné čitelnosti formátu JSON byla využita externí webová aplikace JSON Hero [67] vyobrazená na obrázku 24. Do této aplikace byl nahrán obsah souboru, čímž se data transformovaly do přehledných interaktivních karet. Následně v sekci *vulnerabilities* se přehledně vyobrazily jednotlivé nálezy. Na obrázku 24 lze vidět výstup transformovaný pomocí této aplikace. U každé chyby jsou nejdůležitější tři klíčové parametry:

* **Severity** **(Závažnost)** – Kritičnost daného nálezu.
* **Message (Zpráva)** – Srozumitelný popis, o jaký typ útoku se jedná.
* **File (Soubor)** – Přesná cesta v adresářové struktuře aplikace Juice Shop, která určuje konkrétní řádek kódu, jenž je potřeba opravit.



Obrázek 25 Transformovaná výsledná data pomocí aplikace JSON Hero

V této webové aplikaci následně lze vidět, že nástroj GitLab SAST nalezl 91 zranitelností. Zranitelnosti byly seřazeny od nejkritičtější po nejméně kritické.

Analyzátor nalezl 31 kritických zranitelností, jednou z nich je například zranitelnost typu NoSQL Injection. Skener identifikoval nebezpečné použití funkce *findOne()*, která zpracovává neošetřený vstup od uživatele. Tuto zranitelnost může útočník zneužít a obejít proces autentizace nebo získat přístup k citlivým datům bez znalosti přístupových údajů pomocí speciálních databázových operátorů. V prostředí NoSQL databází je toto riziko srovnatelné s klasickým SQL Injection.

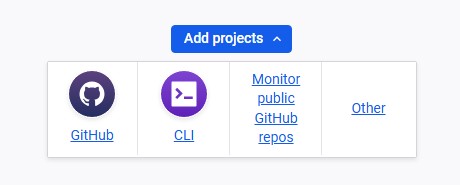
Dále se zde vyskytuje 5 zranitelností s vysokou (high) závažností dle CVSS a příklad je zranitelnost použití natvrdo zapsaných přihlašovacích údajů. Je to jedna z nejčastějších začátečnických chyb ve vývoji.

Nalezeny byly i zranitelnosti se střední (medium) závažností v počtu 55. Nalezena je například zranitelnost „Unchecked input for loop condition“, což znamená, že aplikace provádí cyklus nad daty, která poslal uživatel, aniž by zkontrolovala jejich množství nebo velikost. Útočník může do aplikace poslat záměrně naddimenzovaný vstup. Zpracování takového vstupu vyčerpá systémové zdroje serveru, což vede k útoku typu Denial of Service (DoS). Aplikace se stane pro legitimní uživatele nedostupnou nebo dojde k jejímu pádu.

S nízkou (low) závažností analýza nalezla pouze jednu zranitelnost, a to zranitelnost „Eval Injection“. Jedná se o místo, kde aplikace dynamicky importuje moduly pomocí funkce *require*, přičemž název modulu není pevně definován, ale je tvořen proměnnou. Tato praktika umožňuje útočníkovi manipulovat s cestou k importovaným souborům. V nejhorším případě může útočník donutit aplikaci načíst a spustit libovolný škodlivý kód, který se mu podaří na server nahrát, nebo načíst citlivé systémové soubory.

### Snyk

Nástroj Snyk je možné využít v cloudu, a proto je nejprve nutné založení účtu nebo přihlášení do webové aplikace Snyk. Přihlášení je možné pomocí účtu GitHub, což bylo v tomto případě zvoleno jako nejjednodušší řešení. Přihlášení prostřednictvím GitHubu umožnilo do aplikace jednoduše nahrát zranitelnou webovou aplikaci Juice Shop. Nahrání probíhalo následovně – v levé sekci se kliknulo na záložku „Projects“, kde se vyskytovalo tlačítko „Add projects“ a po kliknutí na toto tlačítko se zobrazila nabídka, která je vyobrazená na obrázku Obrázek 25 a obsahuje odkazy na *GitHub*, *CLI*, *Monitor public GitHub repos* a *Other*.



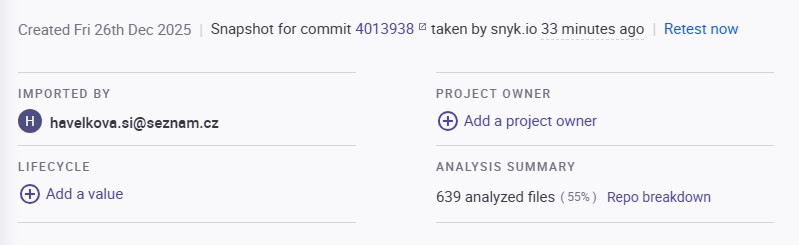
Obrázek 26 Nabídka přidání projektu na platformě Snyk

Pro tento projekt byla zvolena první možnost, tedy *GitHub*. Po zvolení této možnosti se zobrazily všechny repozitáře daného účtu, včetně těch soukromých. Z těchto repozitářů byl vybrán Juice Shop a hned po přidání se jako projekt v aplikaci Snyk zanalyzoval. Analýza trvala v desítkách sekundách, tedy byla velmi rychlá. Po rozkliknuti názvu tohoto projektu se objevila nabídka, včetně možnosti „Code analysis“. Již v této nabídce šlo vyčíst, kolik zranitelností a jakého stupně aplikace uchovává, jak lze vidět na obrázku 26.



Obrázek 27 Počet zranitelností jednotlivých stupňů Snyk analýzy

Po kliknuti na možnost „Code analysis“ se zobrazily podrobnější údaje analýzy. Hned v úvodu se zobrazovaly informace, jako například, kdy byl projekt vytvořen, kdo projekt naimportoval, kolik souborů bylo procházeno. Některé informace chyběly, přičemž se daly dodatečně doplnit, jako například vlastník projektu, prostředí a další. Část zobrazených základních informací lze vidět na obrázku 27.



Obrázek 28 Základní informace projektu a analýzy na platformě Snyk

Hned pod těmito základními informacemi se zobrazují již jednotlivé zranitelnosti. Aplikace umožňuje výsledné zranitelnosti filtrovat podle stupně závažnosti, skóre priority, programovacího jazyka a v poslední řadě podle typu zranitelnosti.

Jak již bylo vidět na obrázku 26, analyzátor nalezl 27 zranitelností zařazených do vysoké závažnosti, 20 do závažnosti střední a 220 do nízké závažnosti. Tyto údaje byly dohledatelné i ve filtrech zranitelností, jak je možné vidět na obrázku 28.



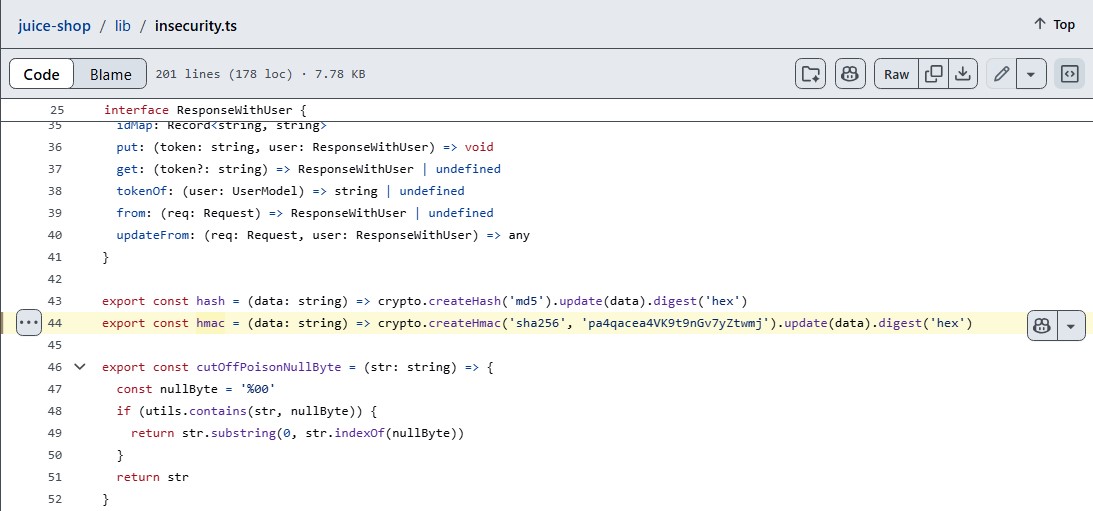
Obrázek 29 Filtr zranitelností podle skóre závažnosti v aplikaci Snyk

Mezi zranitelnosti s vysokou závažností se vyskytovala zranitelnost typu „Hardcoded Secret“ neboli doslova přeloženo „zakódované tajemství“. Tato chyba, klasifikovaná pod označením CWE-547, spočívá v přímém vložení citlivých údajů do zdrojového kódu aplikace,přičemž ve zranitelné aplikaci Juice Shop se vyskytuje podle této analýzy 4krát. V analyzovaném souboru *insecurity.ts* na řádku 23 byl detekován soukromý šifrovací klíč RSA, který byl do proměnné *privateKey* přiřazen jako prostý textový řetězec. Tyto informace se dají zjistit pouze z náhledu, který lze vidět na obrázku 29.



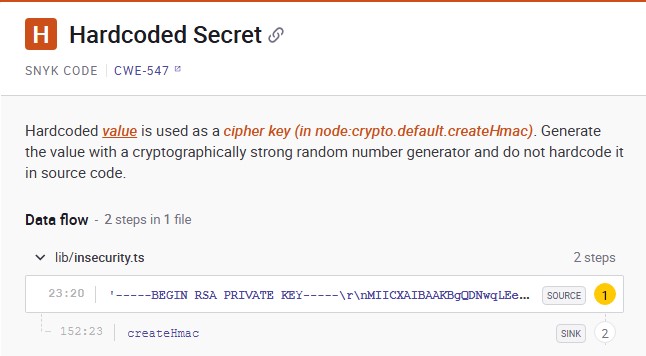
Obrázek 30 Ukázka stručného výsledku zranitelnosti z analýzy Snyk

Na obrázku 29 lze také vidět, že náhled obsahuje odkaz na GitHub. Po otevření odkazu se zobrazil soubor, který zranitelnost obsahuje, včetně zvýrazněného řádku k opravě, viz obrázek 30.



Obrázek 31 Zvýrazněný řádek se zranitelností v aplikaci GitHub

Aplikace nabízí také podrobnější přehled, jak zranitelnost postupuje strukturou aplikace. Na obrázku 31 lze vidět levou část tohoto přehledu, která vyobrazuje panel „Data flow“, který je pro proces statické analýzy klíčový. Tento panel mapuje cestu citlivého údaje ve dvou krocích. První krok je bod „Source“ na řádku 23, kde je definována proměnná *privateKey* obsahující statický řetězec RSA klíče. Druhým bodem je „Sink“ na řádku 152, což je místo, kde jsou tato data předána nebezpečné funkci. Tímto Snyk poukazuje na to, že se nejedná pouze o zapomenutý komentář, ale o aktivně využívaný kód, který přímo ovlivňuje bezpečnost programu.



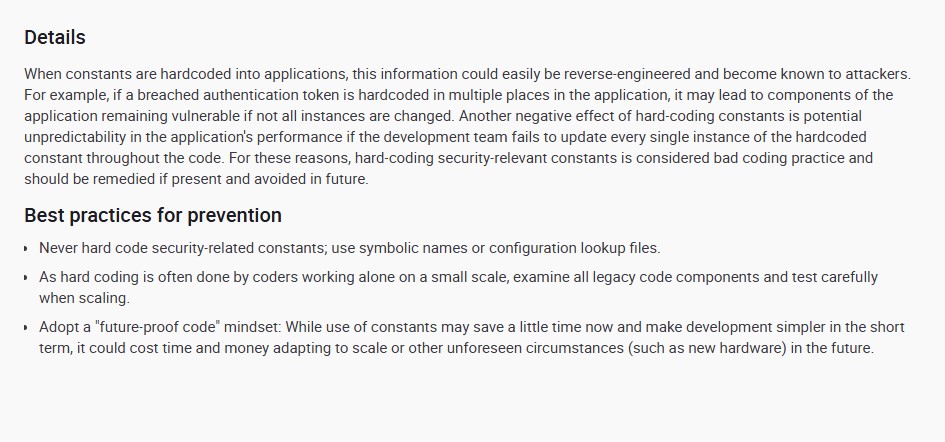
Obrázek 32 Tok dat citlivého údaje z analýzy Snyk

Pravá část obsahuje samotný zdrojový kód souboru *insecurity.ts*, kde je žlutým zvýrazněním označen kritický řádek, viz obrázek 32.



Obrázek 33 Kód se zvýrazněným kritickým řádkem

Tento detailnější přehled nabízí i sekci „Fix analysis“, která slouží k edukaci vývojáře a poskytuje konkrétní příklady, jak zranitelnost daného typu správně opravit. Tato sekce je rozdělena na teoretický výklad na levé části sekce, jak lze vidět na obrázku 33 a praktickou ukázku kódu na pravé části obrázku 34.



Obrázek 34 Teoretický výklad možné opravy chyby



Obrázek 35 Ukázka kódu s opravami

V levé části, tedy v části s teoretickým výkladem, byly podrobně rozvedeny důvody, proč je ukládání konstant přímo do aplikace nebezpečné – zmiňuje se zde zejména riziko reverzního inženýrství. Snyk zde definuje i tzv. „Best practices for prevention“, což jsou osvědčené postupy pro prevenci, mezi které patří používání symbolických jmen nebo konfiguračních souborů namísto přímých hodnot.

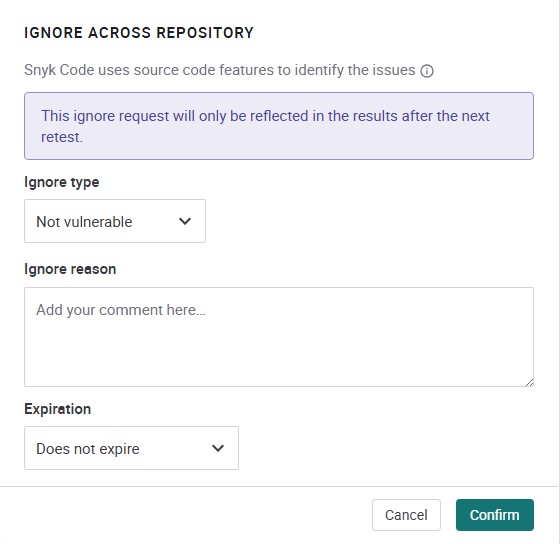
Nejpřínosnější částí pro praktické řešení problému je pravá část s nadpisem „Example fixes“. Zobrazuje reálnou ukázku opravy. Pomocí barevného zvýraznění ukazuje rozdíl mezi původním a opraveným stavem. Červeně zvýrazněné řádky, předcházené znaménkem „-“, značí bezpečný kód, kde bylo tajemství, například *APP\_SECRET*, zapsáno jako statický řetězec. A zeleně zvýrazněné řádky, předcházené znaménkem „+“, demonstrují správné řešení.

Ve spodní části této sekce se vyskytuje tlačítko s nápisem „Ignore across repository“. Toto tlačítko slouží pro trvalé skrytí chyby ze všech budoucích reportů, a to napříč všemi větvemi projektu i různými rozhraními. Důvodů proč vývojář může tuto funkci využít je mnoho a jednou z nich může být například falešně pozitivní nález. Pokud vývojář potřebuje nějakou chybu ignorovat, klikne na toto tlačítko a zobrazí se formulář, viz obrázek 35. Vývojář musí uvést „Ignore type“, který definuje důvod, proč byla daná chyba vyřazena z aktivního sledování. Na výběr jsou tři možnosti:

* **Not vulnerable** (Není zranitelné) – Tato možnost se volí v případě falešně pozitivního nálezu.
* **Ignore temporarily** (Ignorovat dočasně) – Používá se pro odložení nápravy. Tato volba je užitečná v situacích, kdy tým o chybě ví, ale plánuje ji opravit až v příštím vývojovém cyklu.
* **Won’t Fix** (Nebude opraveno) – Tato kategorie označuje akceptované riziko. Volí se u nálezů, jejichž oprava by byla příliš nákladná nebo technicky nerealizovatelná vzhledem k minimálnímu reálnému dopadu na bezpečnost.

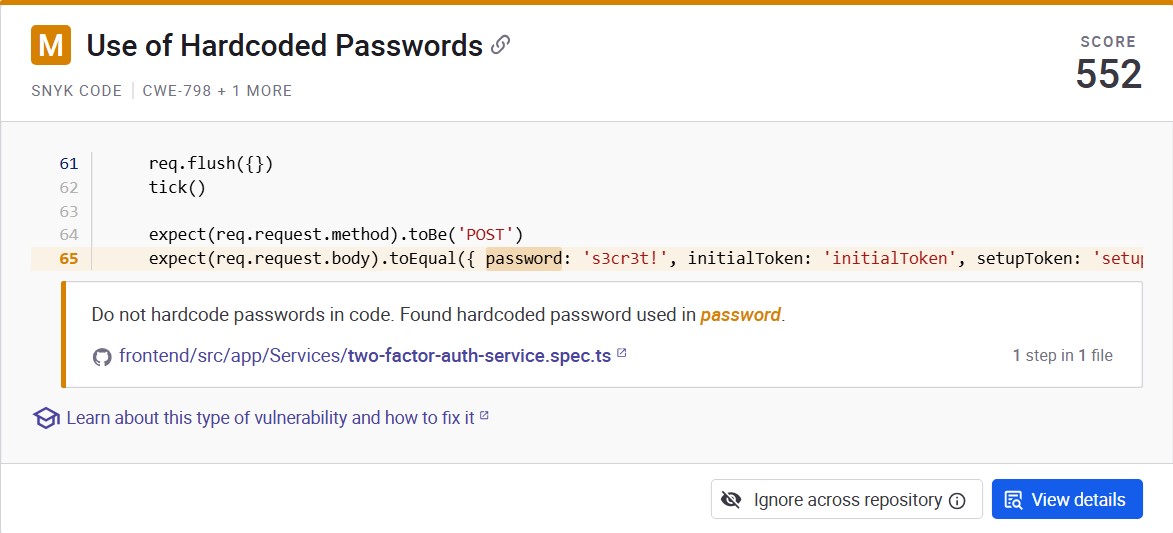
Dále je dobré uvést podrobné vysvětlení svého kroku do pole „Ignore reason“. Tento komentář je klíčový pro ostatní členy týmu, kteří by se mohli k danému souboru v budoucnu vrátit, aby rozuměli, proč byla zranitelnost vyřazena z aktivního monitoringu.

A jako poslední věc k uvedení je zde nastavení „Expiration“. Snyk umožňuje, zda má ignorování platit trvale – možnost „Does not expire“, nebo zda má být časově omezeno – možnost „Custom expiration“.



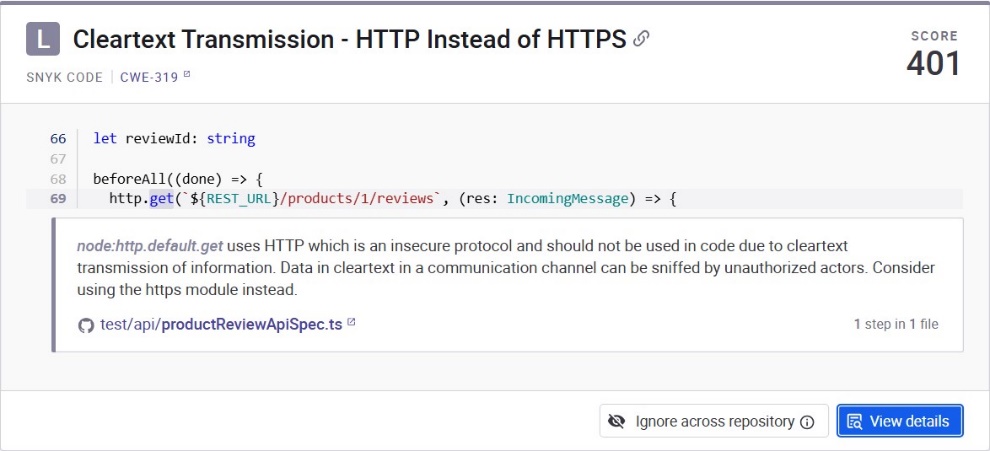
Obrázek 36 Formulář pro ignoraci identifikované zranitelnosti

Jednou z identifikovaných chyb touto analýzou se střední závažnosti byla například zranitelnost „Use of Hardcoded Passwords“, klasifikovaná jako CWE-798.Její náhled lze vidět na obrázku 36. Snyk tomuto nálezu přiřadil skóre 552, což jej řadí do kategorie střední až vysoké závažnosti. Podstatou problému je přítomnost testovacího hesla „s3cr3t!“ přímo ve zdrojovém kódu aplikace. Celkově se tato zranitelnost v aplikaci vyskytuje nejčastěji. Ve filtrech byl u této zranitelnosti uveden počet 152.



Obrázek 37 Ukázka zranitelnosti se střední závažností

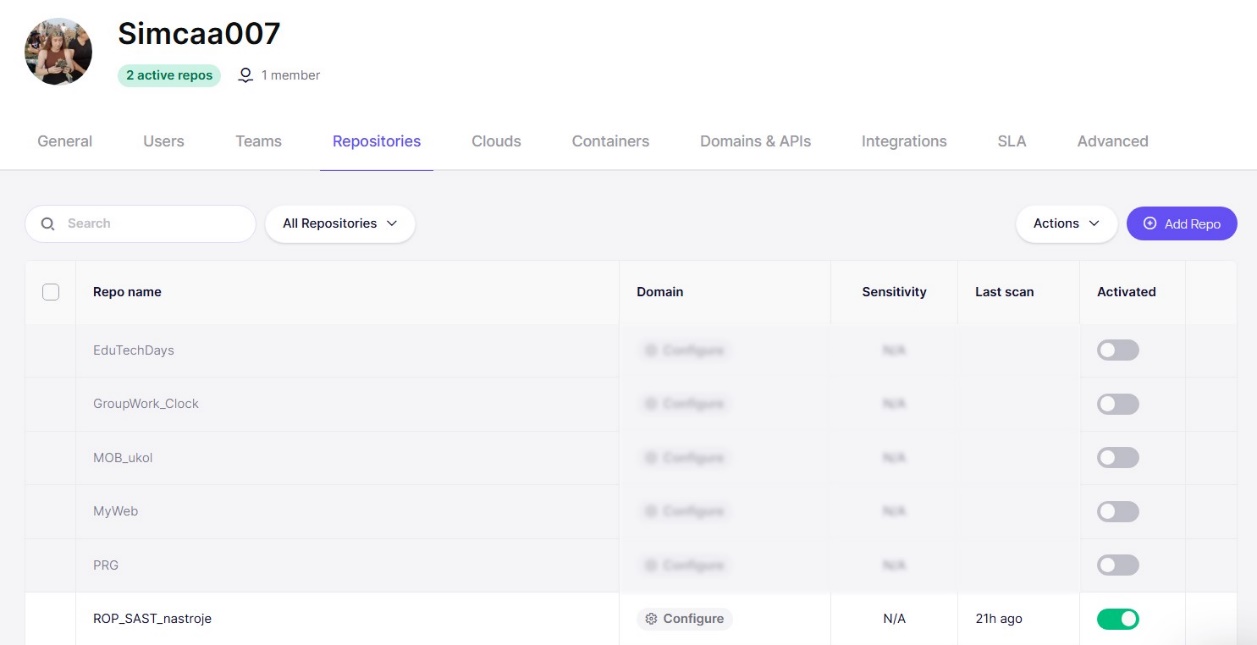
Ze zranitelností nízké závažnosti je například nález pod názvem „Cleartext Transmission – HTTP Instead of HTTPS“, evidovaný pod označením CWE-319. Podstatou zranitelnosti je použití nebezpečného protokolu HTTP namísto jeho šifrované varianty HTTPS při provádění síťového požadavku. Tato zranitelnost je náchylná na útoky typu Man-in-the-Middle, kdy útočník nacházející se na stejné síti může pomocí specializovaných nástrojů pro odposlech tento provoz neautorizovaně zachytit a přečíst. Tento nález je vyobrazen na obrázku 37.



Obrázek 38 Ukázka zranitelnosti nízkého stupně

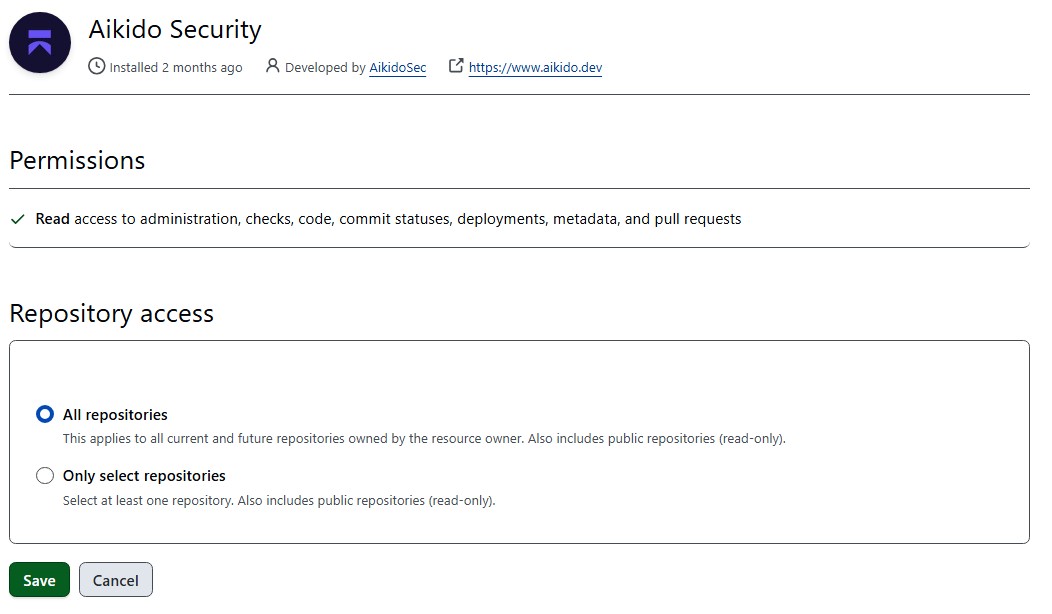
### Aikido

Dalším nástrojem, který lze používat v cloudu, je Aikido. Nejprve proběhlo přihlášení opět přes GitHub*.* Pro provedení analýzy bylo nejdříve potřeba přidat repozitáře z GitHubu, které měly být zanalyzovány. V levé liště je záložka „Repositories“, kde se po kliknutí na tuto záložku zobrazily veškeré repozitáře, které byly doposud přidány do této aplikace. Pro přidání repozitáře vybrané zranitelné aplikace tato záložka obsahovala tlačítko „Manage Repos“. Toto tlačítko zobrazilo všechny repozitáře účtu, jak lze vidět na obrázku 38.



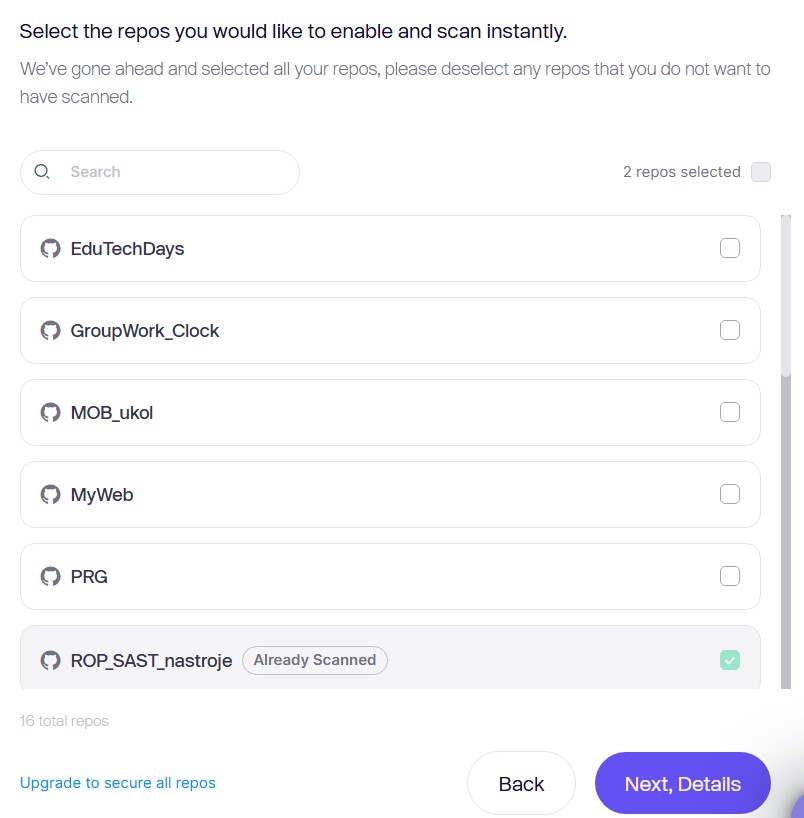
Obrázek 39 Výpis repozitářů v aplikaci Aikido

Následně bylo stisknuto tlačítko „Add Repo“, díky kterému proběhlo přesměrování na stránku GitHub, konkrétně do nastavení „Repository access“, viz obrázek 39.



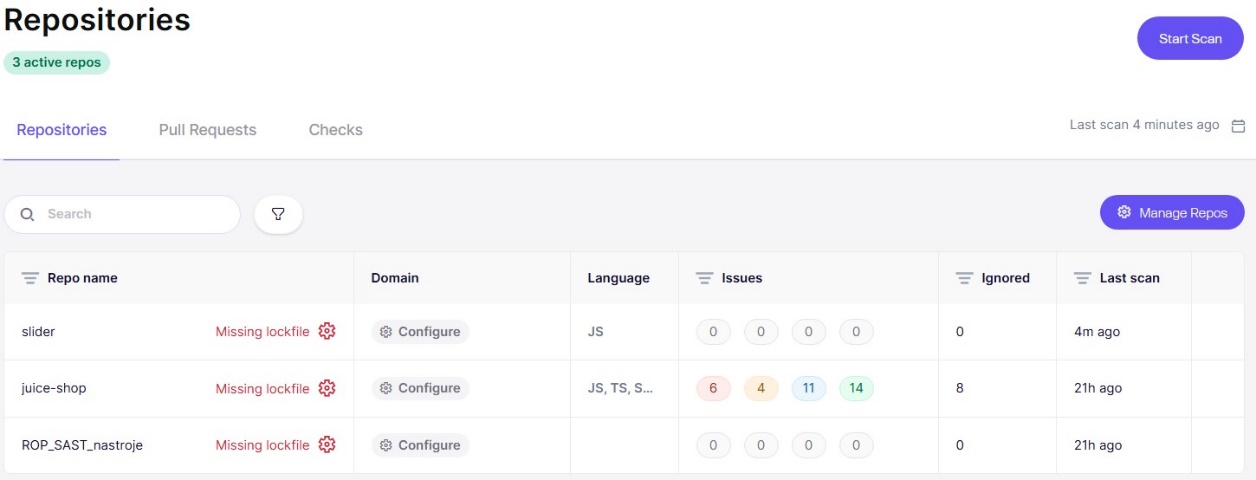
Obrázek 40 Nastavení pro přidání repozitáře z GitHubu do aplikace Aikido

Na obrázku 40 lze vidět, že po vybrání volby „All repositories“ se zobrazila stránka, kde byly vypsány repozitáře, a aplikace Aikido následně požádala o vybrání repozitářů ke statické analýze.



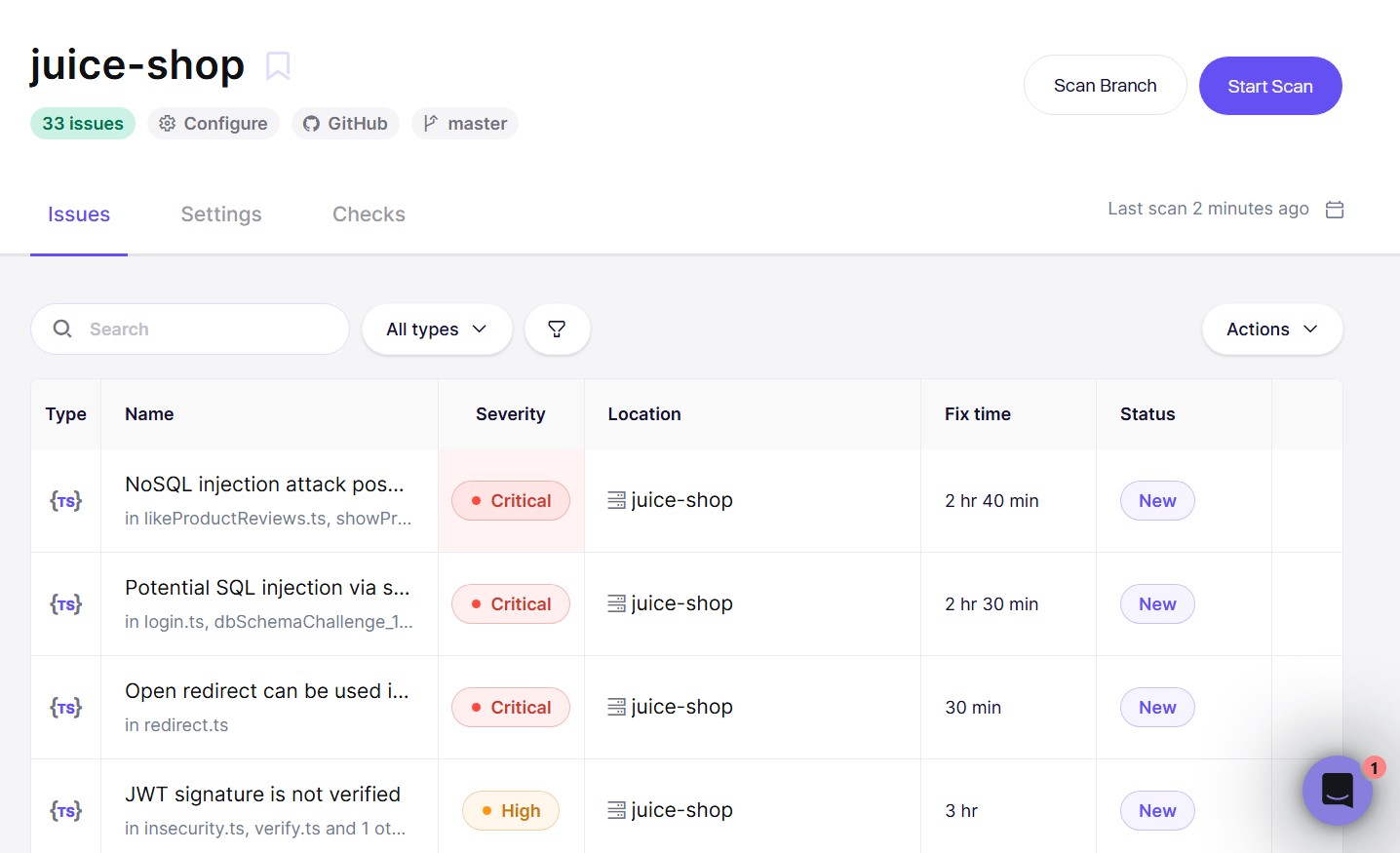
Obrázek 41 Výběr repozitářů, které mohou být skenovány aplikací Aikido

Byl tedy vybrán repozitář vybrané zranitelné aplikace a kliknuto na tlačítko „Next, Details“. To vedlo k přidání repozitáře a spuštění statické analýzy aplikace. Pro zobrazení reportů bylo opět klinuto na záložku „Repositories“ v levé části. Tím se zobrazily všechny repozitáře, které byly zvoleny, včetně zranitelné aplikace Jucie Shop, což lze vidět na obrázku 41.



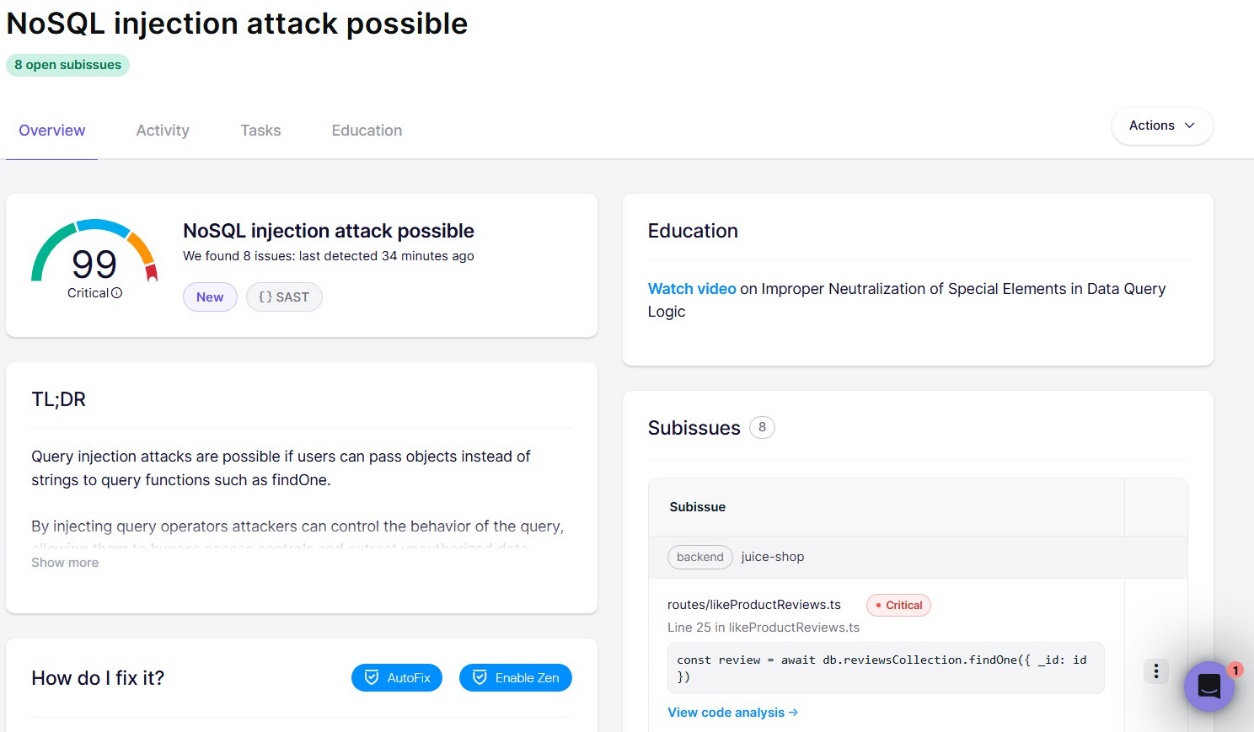
Obrázek 42 Repozitáře a jejich provedená analýza pomocí nástroje Aikido

Na obrázku 41 lze také vidět, že aplikace upozorňuje na „Missing lockfile“. Analýza tedy sice proběhla, avšak pouze částečně, jelikož nebyla provedena analýza knihoven, které zranitelná aplikace obsahuje. Aby mohla být dokončena kompletní analýza, byl přidán soubor *package-lock.json* přímo do repozitáře v aplikaci GtiHub. Tento soubor byl stažen a přidán do repozitáře od vývojáře *bkimminich* z jeho repozitáře *juice-shop* [68]. Jakmile aplikace Aikido zaregistrovala změnu, analýzu spustila znovu. Tato aplikace umožňuje zobrazit pouze vybrané zranitelnosti a jejich podrobný popis. Na obrázku 42 lze vidět část reportů z analýzy nad webovou aplikací Juice Shop. Také je na obrázku 42 možné vidět, že analýza zachytila celkem 33 zranitelností.



Obrázek 43 Část výpisu reportů analýzy Aikido

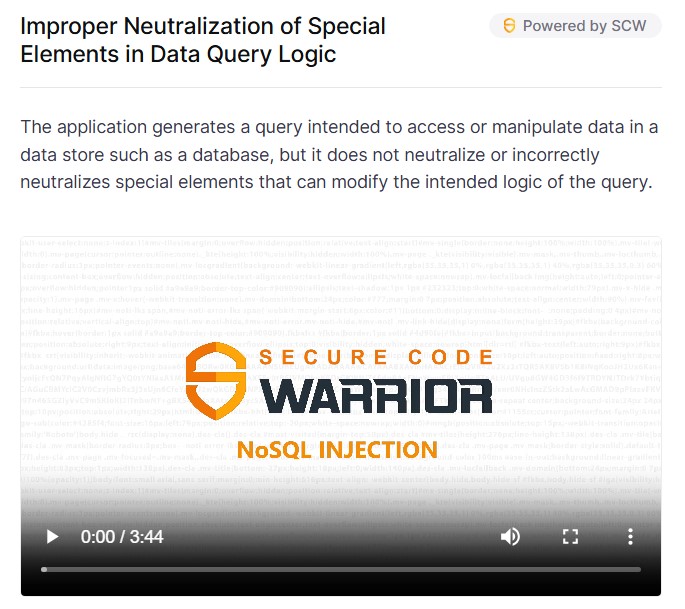
Jak již bylo uvedeno, analyzátor nalezl 33 typů zranitelností, ale pro tuto práci jsou důležité pouze nálezy statické analýzy, tedy 9 typů zranitelností. Z těchto nalezených typů zranitelností byly 4 s kritickou závažností, přičemž jednou z nich byla zranitelnost s názvem „NoSQL injection attack possible“, která dosáhla nejvyššího stupně závažnosti skóre 99. Tento report lze vidět na obrázku 43.



Obrázek 44 Zranitelnost kritické závažnosti objevená analyzátorem Aikido

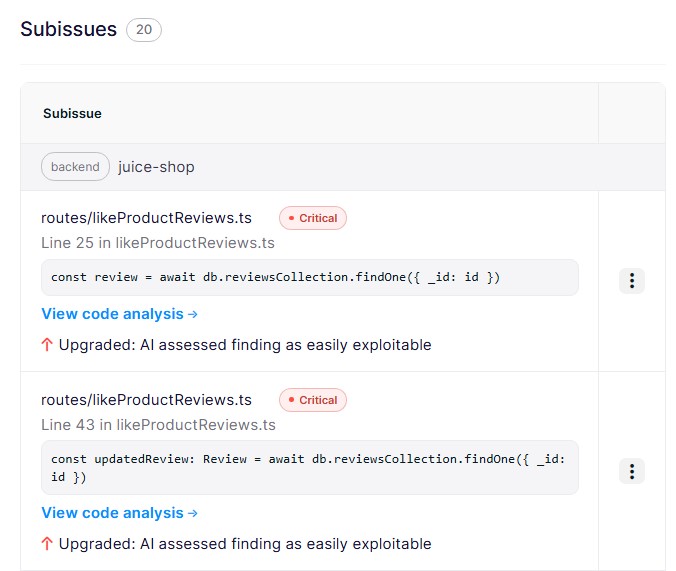
V sekci „TL;DR“, což je rychlé shrnutí bezpečnostního problému nebo zranitelnosti, je popsané, že pokud aplikace neprovádí dostatečnou validaci uživatelských vstupů, může dojít k útoku typu query injection. Tento útok umožňuje útočníkovi manipulovat s databázovým dotazem, obejít přístupová omezení a získat neoprávněný přístup k datům jiných uživatelů.

Dále zde byla sekce „Education“, která obsahuje výukové video platformy Secure Code Warrior, které vysvětluje princip této zranitelnosti, možné způsoby zneužití a doporučené postupy pro její odstranění. Tato sekce lze vidět na obrázku 44.



Obrázek 45 Sekce „Education“ v analyzátoru Aikido

„Subissues“ byla další sekce, kterou nalezený typ zranitelnosti obsahoval. Tato sekce obsahovala dílčí zranitelnosti, které byly identifikovány při statické analýze. Tyto zranitelnosti se vyskytují v různých souborech aplikace, přičemž je vždy uveden konkrétní řádek kódu, na kterém byl problém detekován. V této sekci byly uvedeny veškeré nalezené zranitelnosti, včetně úrovně závažnosti. Tento typ závažnosti, tedy „NoSQL injection attack possible“, obsahoval 4 konkrétní zranitelnosti kritické úrovně závažnosti, 4 zranitelnosti střední závažnosti a 12 ignorovaných zranitelností. Všechny ignorované zranitelnosti jsou popsány nástrojem jako falešně pozitivní nálezy. Tyto ignorované nálezy lze přestat ignorovat, pokud vývojáři s tímto zařazením nesouhlasí. Část této sekce lze vidět na obrázku 45.



Obrázek 46 Sekce „Subissues“ v nalezené zranitelnosti nástroje Aikido

Jak lze vidět na obrázku 45, jedna ze zranitelností sekce „Subissues“ se nachází na řádku 25. Konkrétní řádek je vypsán,kde tento kód provádí databázový dotaz nad kolekcí *reviewsCollection,* přičemž hodnota *id* je použita přímo v dotazu bez předchozí validace nebo ošetření. Součástí nálezu je také informace „Upgraded: AI assessed finding as easily exploitable“, která znamená, že nástroj s využitím umělé inteligence vyhodnotil tuto zranitelnost jako snadno zneužitelnou, a proto jí byla přiřazena vysoká závažnost. Odkaz „View code analysis“ umožňuje zobrazit detailní rozbor problému a doporučení pro jeho odstranění. Po kliknutí na tento odkaz se zobrazí podrobnosti, jak lze vidět na obrázcích 46. Na obrázku 46 lze vidět, že aplikace vypsala kód, ve kterém se zranitelnost nachází, a konkrétní řádek zvýraznila žlutou barvou.



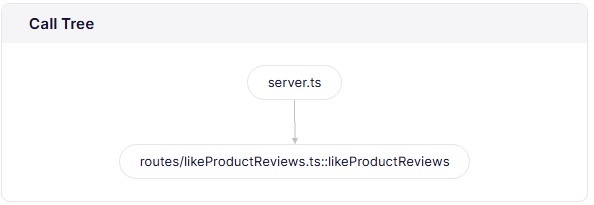
Obrázek 47 Ukázka kódu detekované zranitelnosti NoSQL injection

„AI Autotriage Summary“ byla další část, kterou detailnější rozbor obsahoval, a lze ji vidět na obrázku 47. Skener uvádí, že uživatelsky kontrolovaný vstup *req.body.id* je předáván do databázových dotazů MongoDB bez předchozí sanitizace, což umožňuje NoSQL injection. Také je zde uvedeno, že je nutná okamžitá oprava.



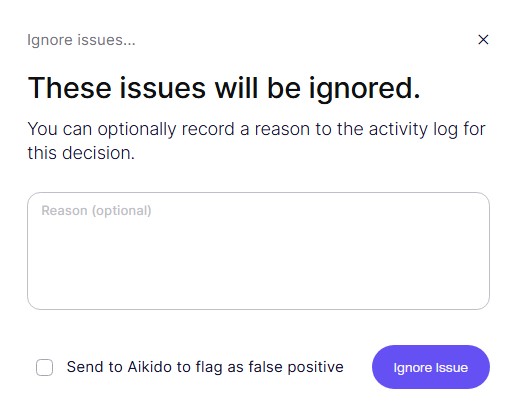
Obrázek 48 AI souhrn ke zranitelnosti v aplikaci Aikido

Poslední informaci, která se v tomto detailnějším rozboru zobrazuje, nese název „Call Tree“. Sekce „Call Tree“ představuje grafické znázornění toku dat v rámci zdrojového kódu aplikace. V kontextu nalezené zranitelnosti NoSQL Injection slouží tento strom k vizualizaci cesty, kterou urazí nebezpečný vstup od okamžiku přijetí systémem až po jeho kritické využití v databázovém dotazu. Tento strom je vyobrazen na obrázku 48.



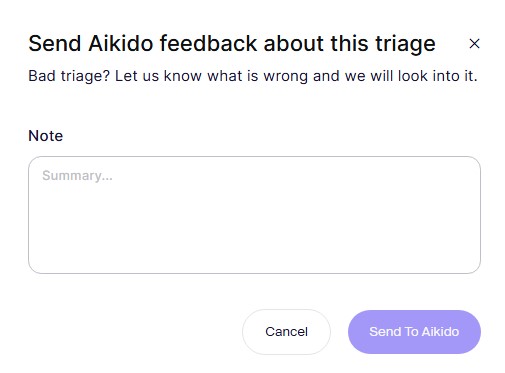
Obrázek 49 Grafické znázornění toku dat zranitelnosti v aplikaci Aikido

U předešlého nástroje GitLab byla k dispozici možnost ignorace zranitelnosti, přičemž tato možnost je dostupná i zde, a to v pravé části detailnějšího rozboru. Pokud je z nějakého důvodu nutné zranitelnost ignorovat, zobrazí se formulář, viz obrázek 49. Na rozdíl od nástroje GitLab vyžaduje Aikido pouze popis důvodu, proč má být zranitelnost ignorována, a nabízí také možnost zaškrtnutí checkboxu, že se jedná   
o falešně pozitivní nález.



Obrázek 50 Ignorace nalezené zranitelnosti v nástroji Aikido

Funkci poslání zpětné vazby ovšem Snyk oproti nástroji Aikido nenabízí. Tento nástroj tuto funkci má, díky čemuž lze ke každé nalezené zranitelnosti odeslat zpětnou vazbu. Tuto funkci lze např. využít v případě, kdy nalezená zranitelnost představuje falešně pozitivní nález. Pro odeslání zpětné vazby nástroj obsahuje formulář, který vyžaduje uvedení důvodu, proč je analýza chybná. Formulář lze vidět na obrázku 50.



Obrázek 51 Formulář pro zpětnou vazbu aplikace Aikido

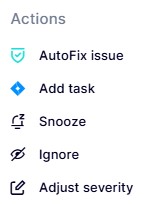
Poslední funkcí je funkce „AutoFix preview“. Tato funkce představuje významný posun v automatizaci bezpečnosti, kdy systém nejen že chybu vyhledá, ale zároveň navrhne konkrétní programový kód k jejímu odstranění. Systém využívá umělou inteligenci k vytvoření tzv. „pacthe“ (záplaty). U návrhu k odstranění této identifikované chyby NoSQL Injection AI vysvětlovalo, že navržená oprava zmírňuje riziko injekce vynucením striktní rovnosti v databázovém poli *\_id* pomocí operátoru *$eq*. Rozhraní pak následně zobrazovalo přehledný rozdíl mezi původním nebezpečným kódem (červeně vybarvený a se symbolem „-“) a novým zabezpečeným kódem (vybarvený zelenou barvou se symbolem „+“).

U návrhu byla uvedena hodnota „Medium Confidence“. To je důležitý údaj pro vývojáře, který říká, že ačkoliv je oprava pravděpodobně správná, je nutná finální lidská kontrola před jejím nasazením. Právě z tohoto důvodu návrh obsahoval i ikonky, zda to byl dobrý návrh nebo špatný. A jako poslední tato možnost obsahovala funkci „Create PR“. Tato možnost znamená, že lze vytvořit Pull Request, tedy systém sám vytvoří novou větev v repozitáři a připraví kód ke schválení, čímž se drasticky zkracuje doba, po kterou je aplikace zranitelná. Zobrazení této funkce pod zranitelností NoSQL Injetion lze vidět na obrázku 51.



Obrázek 52 Funkce „AutoFix preview“ nástroje Aikido

Jak již bylo vidět na obrázku 45, každý náhled zranitelnosti měl v pravé části ikonku tří teček, která obsahovala 5 akcí, viz obrázek 52.

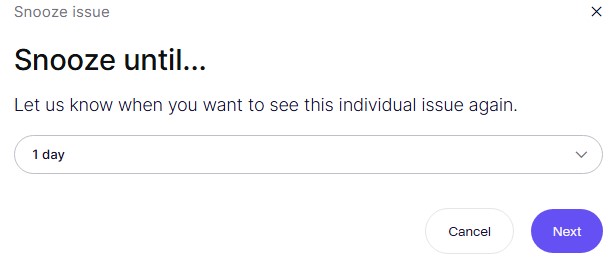


Obrázek 53 Akce pro každou nalezenou zranitelnost v nástroji Aikido

Jako první akce, co u zranitelnosti nástroj nabízel, byla akce „AutoFix issue“, která již byla zmíněna výše, a tedy vygeneruje během několika vteřin doporučení, jak chybu opravit. Po zvolení této akce se zobrazil stejný náhled, jak lze vidět na obrázku 51.

„Add task“ byla další z 5 možných akcí, která v platformě Aikido transformuje teoretické nálezy bezpečnostního skenu do praktických kroků v rámci životního cyklu vývoje software (SDLC). Umožňuje bezproblémovou synchronizaci s externími nástroji pro správu úkolů, čímž zajišťuje, že kritické zranitelnosti nezůstanou pouze v reportu, ale budou systémově přiděleny vývojářům k nápravě a jejich stav bude monitorován až do úplného vyřešení. Tato funkce nebyla prakticky vyzkoušena, protože vyžaduje placenou verzi aplikace.

V rámci správy bezpečnostních incidentů v nástroji Aikido je důležitým prvkem funkce „Snooze issue“. Tato funkce umožňuje dočasně skrýt konkrétní zranitelnost z aktivního seznamu úkolů na předem definované časové období. Tato funkce je přístupná i pro neplacenou verzi, a tedy po zvolení této funkce se zobrazil jednoduchý formulář. Pro skrytí stačilo pouze vybrat, pro jak dlouhou dobu má být zranitelnost skryta. V tomto případě byla zvolena možnost jednoho dne, jak lze vidět na obrázku 53, protože se jedná pouze o testování nástroje.



Obrázek 54 První formulář funkce pro skrytí zranitelnosti na určitou dobu

Po kliknutí na tlačítko „Next“ se vyobrazil druhý formulář s datem, do kdy bude zranitelnost skryta. K vyplnění tento formulář požadoval popis, proč má být zranitelnost skryta, viz obrázek 54.



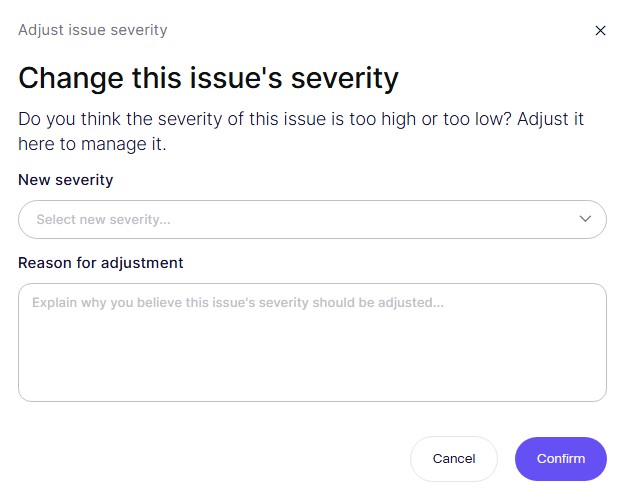
Obrázek 55 Druhý formulář funkce pro skrytí zranitelnosti na určitou dobu

Při práci s automatizovanými skenery se běžně vyskytují situace, kdy je nutné nález trvale vyřadit z monitorování. K tomuto účelu slouží v platformě Aikido funkce „Ignore“ jako další z funkcí pod třemi tečkami. Tato funkce byla již zmíněna, ovšem pod tímto odkazem se chovala trochu jinak. Přes tuto funkci bylo na výběr ze tří možností:

* **Only this issue** (Pouze tento výskyt) – Ignorovat bude pouze jednu konkrétní zranitelnost na přesně daném řádku kódu a ostatní podobné chyby v projektu zůstanou aktivní.
* **Ignore by path** (Ignorování podle cesty) – Tato volba vyřadí všechny stávající i budoucí nálezy v konkrétním souboru nebo složce.
* **Disable code rule** (Vypnutí pravidla) – Jedná se o nejširší nastavení, které zcela deaktivuje dané detekční pravidlo pro celý projekt.

Po zvolení jedné z těchto tří možností se opět ukázal stejný formulář, který lze vidět na obrázku 49. Tedy pro dokončení ignorování zranitelnosti už bylo pouze potřeba uvést důvod rozhodnutí pro ignoraci.

Poslední akcí, která byla možná provést pod ikonkou tří teček, byla akce „Adjust severity“. Bezpečnostní nástroje přiřazují automaticky každé zranitelnosti určitou úroveň závažnosti na základě obecných standardů. V reálném provozu však může být dopad stejné chyby v různých částech aplikace odlišný. A právě tato funkce slouží k nápravě těchto rozdílů. Pro změnění úrovně závažnosti se zobrazil opět jednoduchý a přehledný formulář, který obsahoval pouze dvě komponenty, jež uživatel musel vyplnit/zvolit. Nejdříve musela být zvolena pravdivá úroveň závažnosti dle uvážení, a následně musel být uveden důvod úpravy. Na obrázku 55 je vyobrazen tento formulář.



Obrázek 56 Formulář pro změnu úrovně závažnosti nalezené zranitelnosti

S vysokou závažností byl nalezen pouze jeden typ zranitelnosti, a to „Potential file inclusion attack via reading file“, přeloženo jako „Potenciální útok zahrnutím souboru prostřednictvím čtení souboru“. Pokud útočník dokáže ovládat vstupní údaje vedoucí do funkce *ReadFile*, může být schopen číst citlivé soubory, a s těmito informacemi následně spustit další útoky.

Detailnější rozbor zranitelnosti uvedl, že funkce vytváří cesty souborovému systému z externího argumentu *paths* a výpisů adresářů. K sestavení těchto cest využívá metodu *path.resolve* bez ověření, zda vyřešená cesta zůstává v rámci zamýšlené báze, takže je možné procházení cestou nebo čtení libovolných souborů. Tento popis poskytnutý nástrojem byl vygenerován pomocí AI.

Tento detailnější rozbor také obsahoval sekci „Call Tree“ a výpis kódu se zvýrazněným řádkem, kde se zranitelnost nachází, jak tomu bylo i v předešlé zmíněné zranitelnosti. Sekci „Call Tree“ a v ní graf toku dat lze vidět na obrázku 56. Výpis části kódu obsahující zranitelnost vyobrazuje obrázek 57.

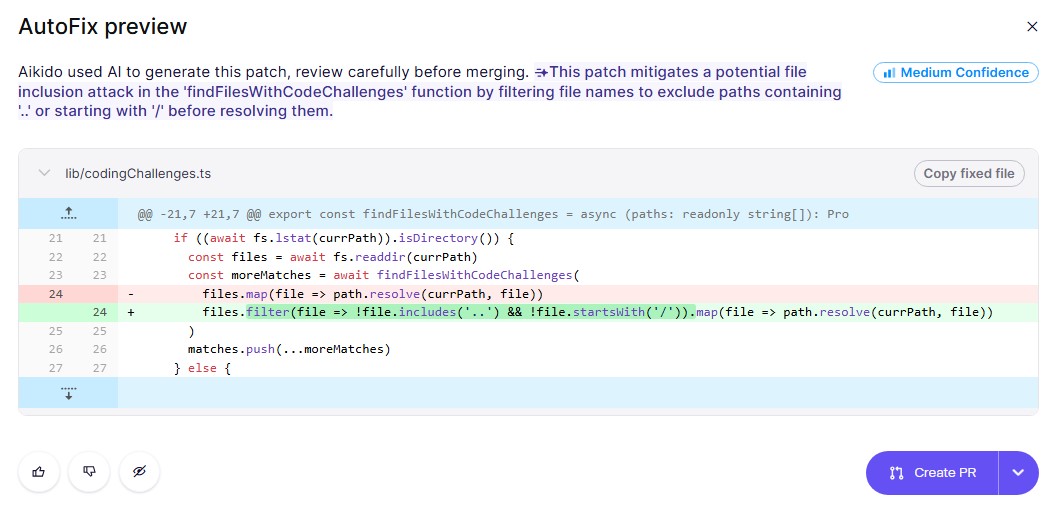


Obrázek 57 Graf toku dat zranitelnosti s vysokou závažností



Obrázek 58 Část kódu obsahující zranitelnost vysoké závažnosti

Tuto zranitelnost analyzátor opravil pomocí AI následujícím způsobem, který lze vidět na obrázku 58. Tato záplata zmírňuje potenciální útok vkládáním souborů do funkce „findFilesWithCodeChallenges“ filtrováním názvů souborů tak, aby se před jejich vyřešením vyloučily cesty obsahující znaky „..“ nebo začínající znakem „/“.



Obrázek 59 Způsob, jakým lze opravit danou zranitelnost

Posledními typy zranitelností jsou ty se střední závažností. Typ zranitelnosti „Potential XSS due to enabling bypassSecurityTrustUrl“ byla jedna z nalezených. Obsahovala 4 konkrétní zranitelnosti a všechny byly střední závažnosti.

Z těchto 4 zranitelností byla například zranitelnost, kde uživatelem ovládaný e-mail je vkládán do HTML pomocí *bypassSecurityTrustHtml*, což umožňuje vysoce rizikové XSS[[7]](#footnote-7). Výpis 3 ukazuje řádek s touto zranitelnou funkcí.

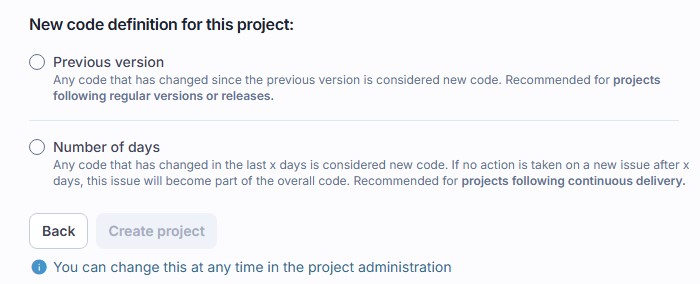
Výpis 3 Řádek obsahující zranitelnou funkci bypassSecurityTrustHtml



Zranitelnosti tohoto typu neobsahovaly funkci AI opravy. Kromě této funkce byly všechny ostatní plně funkční.

### SonarQube

SonarQube byl dalším a zároveň posledním nástrojem, který byl využit v cloudu. Přihlášení tedy proběhlo opět přes účet GitHub. Platforma stejně jako všechny předešlé umožňovala výběr jednotlivých repozitářů k analyzování. Před přidáním projektu nástroj požadoval o definici toho, co se považuje za „nový kód“. Obrázek 59 zachycuje konfigurační krok, který je pro analýzu klíčový, protože určuje, na jakou část programového kódu se mají aplikovat nejpřísnější pravidla kontroly kvality. Tento krok se provádí proto, aby vývojáři mohli prioritně hlídat čistotu aktuálních změn, aniž by byli zahlceni starými nedostatky v historické části aplikace. První možností byla možnost „Previos version“ (Předchozí verzce). Toto nastavení definuje nový kód jako veškeré změny provedené od posledního vydaného sestavení. Druhou variantou byla varianta „Number of days“ (Počet dní), kde se za nový kód považují úpravy za určité časové období, například za posledních 30 dní.



Obrázek 60 Výběr definice nového kódu v SonarQube

Následně v sekci „My Projects“ lze nalézt všechny doposud přidané projekty. Již v této sekci lze u každého projektu vidět náhled výsledků analýzy. Tento náhled obsahuje 5 metrik. Tyto metriky jsou:

* **Security** (Bezpečnost) – Tento údaj ukazuje počet nalezených bezpečnostních chyb.
* **Reliability** (Spolehlivost) – Jedná se o spolehlivost, která počítá chyby v logice nazývané „Bugs“.
* **Maintainability** (Udržovatelnost) – Tato hodnota představuje „Code Smells“ – tedy místa, která nejsou vyloženě chybná, ale jsou špatně zapsaná, a budou se v budoucnu těžko udržovat.
* **Hotspots** **Reviewed** (Prověřená riziková místa) – Tato metrika sleduje, kolik citlivých míst v kódu prověřil člověk.
* **Duplications** (Duplikace) – Udává procento duplicitního kódu.

U vybrané zranitelné aplikace Juice Shop bylo zaznamenáno 35 otevřených problémů v sekci „Secuity“, 127 v sekci „Reliability“, 695 v „Maintainability“, 0 % v „Hotspots Reviewes“ a 6,3 % v „Duplications“. Tento náhled lze vidět na obrázku 60.



Obrázek 61 Náhled zranitelností a ostatní metriky v SonarQube

U většiny sledovaných metrik bylo rovněž uvedeno výsledné hodnocení formou známky. Tuto známku systém generuje automaticky na základě nejvyšší dosažené závažnosti nalezených nedostatků v dané kategorii. Sekce „Security“ získala známku „E“, tedy vyskytovala se zde minimálně jedna zranitelnost kritické závažnosti. Známku „C“ získala sekce „Reliability“, kvůli nalezené zranitelnosti se závažností střední. S nejvyšším počtem nálezů byla sekce „Maintainability“, která obdržela známku „A“, což v SonarQube znamená, že je technický dluh do 5 %. U sekce „Htospots Reviewed“ se nacházela známka „E“, protože kód nebyl do té doby manuálně zkontrolován.

Po rozkliknutí sekce „Security“ se zobrazily zranitelnosti, které byly nalezeny. U každé z nich bylo v náhledu uvedeno, jaký stupeň závažnosti daná zranitelnost má, před jakou dobou kód s určitou zranitelností byl naprogramován, stručný popis zranitelností a mnoho dalších informací. Tento náhled lze vidět na obrázku 61.



Obrázek 62 Náhled jedné z nalezených zranitelností v SonarQube

Aplikace dále nabízí různé možnosti filtrování, přičemž tím základním je řazení podle závažnosti. Nálezy jsou tak přehledně rozděleny do kategorií od nejkritičtějších (Blocker, High) až po méně závažné či čistě informativní (Medium, Low, Info) [69].

Dalším filtrem je filtr „Code Attribute“. Jedná se o klíčovou vlastnost, která ovlivňuje dlouhodobou udržitelnost a hodnotu software. Tyto atributy jsou zařazovány do 4 kategorií: „consistency“ (konzistence), „intentionality“ (záměrnost), „adaptability“ (adaptabilita), „responsibility“ (zodpovědnost). V sekci „Security“ byl 1 nález, který patřil do konzistence, 28 nálezů zařazených do záměrnosti a 6 do zodpovědnosti [69].

Neposledním filtrem je „Type“. Pokud kód nevyhovuje nastaveným pravidlům, je mu přiřazen konkrétní typ incidentu. Systém rozlišuje mezi funkčními chybami (bugy), bezpečnostními zranitelnostmi, nedostatky v čistotě kódu (code smells) a citlivými místy v programu (security hotspots), která vyžadují manuální revizi. V této sekci všech 35 nálezů patřily do bezpečnostních zranitelností [69].

„Type Severity“ byl další filtr, který aplikace nabízela. Každé chybě je přiřazena úroveň závažnosti, která říká, jak rychle je potřeba problém řešit. Škála začíná u kategorie „Blocker“ a pojračuje přes „Crtitical“, „Major“, „Minor“ až po informativní nálezy označené jako „Info“. Zde aplikace zobrazovala 29 nálezů s typem „Blocker“ a 6 s typem „Major“ [69].

Dále nástroj rozřazuje nálezy podle jednotlivých zranitelností. V tomto případě v sekci „Security“ bylo nalezeno 10 zranitelností typu SQL Injection, 7 typu Authentication a další.

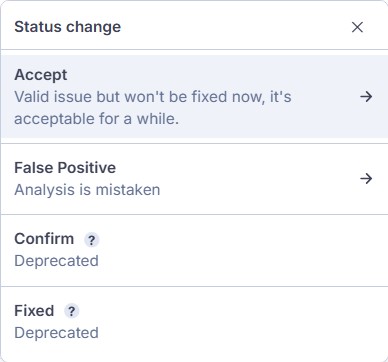
V neposlední řadě umí nástroj rozřazovat zranitelnosti podle toho, v jakém jazyce byly napsány. V této sekci byla většina v jazyce TypeScript.

Filtrů v aplikaci lze najít více, ale posledním důležitým filtrem je filtr podle statusu. Satusy mohou být následující: „Open“, „Confirmed“, „False Positive“, „Accepted“, „Fixed“. V „Security“ byly všechny nálezy v „Open“, protože status musí být zadán ručně.

Již ve výše zmíněném náhledu lze změnit tento status. Možnosti jsou následující:

* **Accept** (Přijmout) – Platný problém, ale nyní nebude opraven, je to na chvíli přijatelné.
* **False** **Positive** (Falešně Pozitivní) – Analýza je chybná.
* **Confirm** (Potvrdit) – Potvrzení, že se opravdu jedná o reálnou hrozbu.
* **Fixed** (Pevný) – Volba, která označuje stav, kdy je zranitelnost považována za odstraněnou.

Pokud by vývojář chtěl zvolit možnost „Accept“ nebo „False Postive“, tak musí dále uvést (jako v každé předešlé platformě) důvod, proč tuto zranitelnost takto označuje. Toto menu pro výběr statusu lze vidět na obrázku 62.

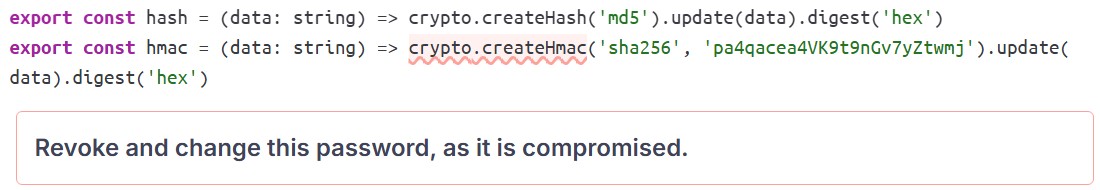


Obrázek 63 Menu pro změnu statusu zranitelnosti v SonarQube

Pro zajištění nápravy nalezených chyb je nezbytné využít funkci přiřazení, která deleguje odpovědnost za konkrétní zranitelnost na vybraného člena týmu. Přidělováním zranitelnosti jednotlivým vývojářům eliminuje chaos při vývoji, neboť jasně určuje, kdo je za opravu zodpovědný.

Jednou z 29 kritických zranitelností bylo použití pevně zakódovaného hesla. Po rozkliknutí této zranitelnosti se zobrazilo menu s 5 možnostmi a první možností byla možnost „Where is the issue?“, tedy kde se problém nachází.

V této možnosti se zobrazila část kódu, kde se zranitelnost nachází, a kritické místo bylo zvýrazněno, jak lze vidět na obrázku 63.

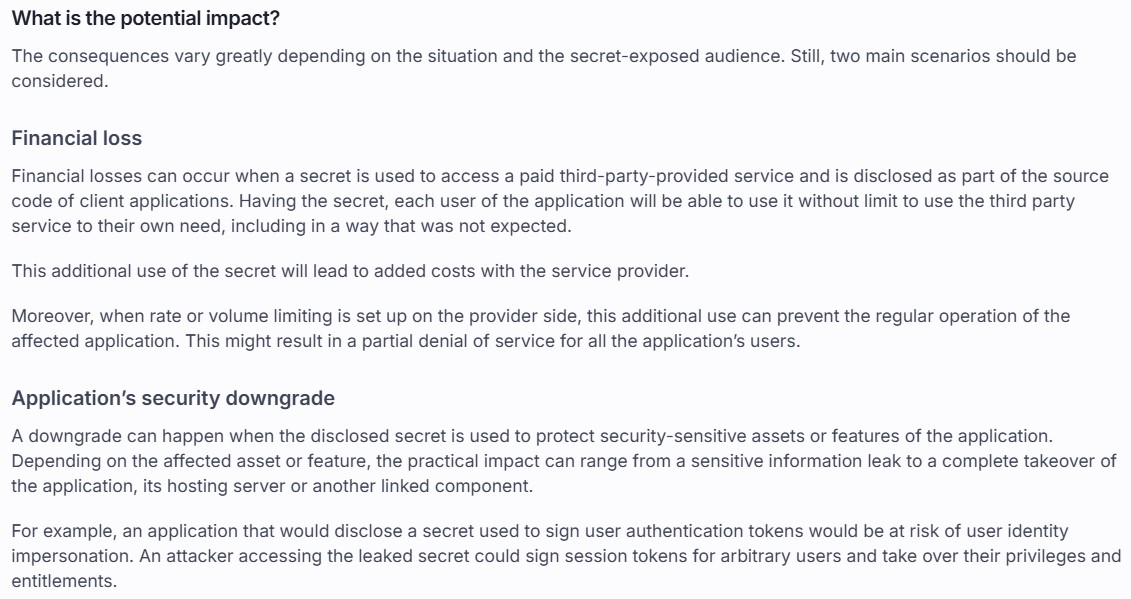


Obrázek 64 Kritická zranitelnost nalezená nástrojem SonarQube

„Why is this an issue?“ v překladu „Proč je to problém?“ byla další možnost, která poskytovala detailní kontext k nalezenému bezpečnostnímu riziku. Jak lze vidět na obrázku 64, text rozvádí potenciální dopady takového pochybení do dvou hlavních kategorií, a to finanční ztráty a snížení bezpečnosti aplikace.

V odstavci finančních ztrát – „Financial loss“ bylo popsáno, že pokud dojde k úniku tajného klíče sloužícího k přístupu ke zpoplatněným službám třetích stran, může kdokoli tyto služby využívat na náklady provozovatele aplikace. Dále bylo uvedeno, že navíc hrozí riziko vyčerpání limitů, což může vést až k částečnému výpadku aplikace pro legitimní uživatele.

 Další část „Application’s security downgrade“ – snížení bezpečnosti aplikace se zaměřuje na to, že únik citlivých dat může vést k široké škále útoků, od úniku informací až po úplné převzetí kontroly nad aplikací nebo hostujícím serverem. Jako příklad bylo uvedeno zneužití klíče pro podepisování autentizačních tokenů, což by útočníkovi umožnilo vydávat se za libovolného uživatele a získat jeho oprávnění.



Obrázek 65 Vysvětlení, proč je daná zranitelnost problém

Nástroj poskytuje návrh, jak lze zranitelnost opravit. Návrh se nacházel ve 3. možnosti z menu. Nejprve zde bylo uvedeno, že pokud dojde k úniku tajného klíče, je nutné tento klíč neprodleně odvolat a odstranit jej ze zdrojového kódu aplikace. Dále nástroj navrhuje prověřit autentizační nebo přístupové logy s cílem odhalit případné neoprávněné či škodlivé využití klíče od okamžiku jeho kompromitace. V neposlední řadě nástroj udával, že je doporučeno využívat specializované úložiště tajných údajů pro generování a ukládání nového tajného klíče.

Kromě teorie platforma zobrazovala příklad nevyhovujícího kódu a vyhovující řešení. U příkladu nevyhovujícího kódu byl řádek červeně zvýrazněn pro poukázání na chybu, to lze vidět na obrázku 65. A u vyhovujícího řešení byl řádek naopak zeleně zvýrazněn, aby byla oprava přehledná, viz obrázek 66.



Obrázek 66 Příklad nevyhovujícího kódu



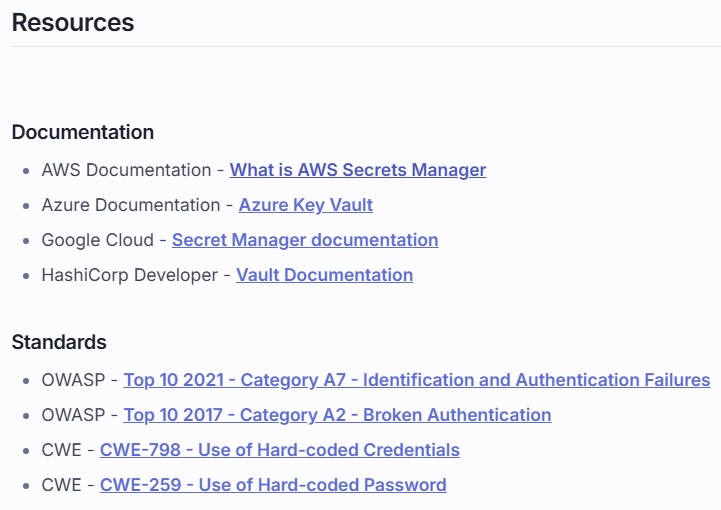
Obrázek 67 Řešení pro příklad nevyhovujícího kódu

Jako předposlední možnost byla „Activity“ neboli aktivita, ve které bylo zobrazeno, kdo a kdy danou zranitelnost vytvořil. Také zde byla možnost přidání komentáře a v textovém poli bylo možné obsah i formátovat, aby mohlo být písmo například tučné. Sekci „Activity“ lze vidět na obrázku 67.



Obrázek 68 Sekce „Activity“ dané zranitelnosti v SonarQube

Jako poslední sekci platforma nabízí „More info“, tedy více informací o dané zranitelnosti, která lze vidět na obrázku 68. Pod nadpisem „Resources“ stála doprovodná část k nalezené zranitelnosti, která poskytovala odkazy na relevantní dokumentaci a bezpečnostní standardy. Jejím cílem bylo umožnit lepší pochopení problému, jeho dopadů a doporučených postupů pro jeho řešení. Další částí byla část „Documentation“, kde byly uvedeny odkazy na oficiální dokumentaci nástrojů pro správu tajných údajů, jako jsou například AWS Secrets Manager, Azure Key Vault, Google Cloud Secret Manager nebo HashiCorp Vaiult. Následující část „Standards“ odkazuje na mezinárodně uznávané bezpečnostní standardy a klasifikace zranitelností, které potvrzují závažnost nalezeného problému. Konkrétně v tomto případě část obsahovala odkaz na OWASP Top 10, dále odkazy o klasifikaci CWE.



Obrázek 69 Nabízené odkazy pro zjištění více informací k dané zranitelnosti

Poslední významnou funkcí, kterou nástroj v detailu zranitelnosti nabízí, je tlačítko „Open in IDE“, což v českém překladu znamená „Otevřít v integrovaném vývojovém prostředí“. Tato funkce má za cíl maximálně usnadnit práci vývojáře tím, že po kliknutí automaticky otevře daný soubor přímo v jeho editoru a kurzor umístí na konkrétní řádek s chybou. Při testování této funkce se však projevilo omezení této funkce, kdy systém zobrazil chybové hlášení „Unable to open the issue in the IDE“. V českém významu to znamená, že „nebylo možné otevřít problém ve vývojovém prostředí“.

Z nálezů zařazených do „Security“ byly nalezeny i zranitelnosti se střední závažností. Přesněji jich nástroj nalezl 6 a z toho 5 jich zařadil i do nízké závažnosti sekce „Maintainability“.

V „Reliability“, tedy chyby, které mohou způsobit nefunkčnost programu, se nacházelo 127 zranitelností, a z toho 8 nálezů zařazených do vysoké závažnosti, 42 do střední závažnosti a 77 do nízké závažnosti. Z těchto 127 nálezů zařazených do „Reliability“ jich bylo spousta zařazených i do „Maintainability“ a ve většině případů pro „Maintainability“ patřily nálezy do nízké závažnosti.

U všech 8 nálezů závažnosti vysoké se objevovaly štítky „async“ a „promise“, které znamenají, že se chyba týká asynchronního programování v JavaScriptu/TypeScriptu. Konkrétně upozorňují na nesprávné zacházení s objekty typu „promise“.

Ve většině nálezů střední závažnosti zranitelnosti dostaly štítek „Code Smell“. Tyto typy problémů mohou v budoucnu vést k závažnějším potížím. Fungují jako varovné signály, jež upozorňují na možné nevhodné programátorské praktiky. Samotné „code smells“ nepředstavují chyby ani bugy, a obvykle nemají přímý vliv na funkčnost software. Poukazují však na slabiny v návrhu aplikace, které mohou zpomalovat další vývoj, nebo zvyšovat riziko vzniku chyb či selhání v budoucnu [70].

Dále zranitelnosti střední závažnosti obsahovaly štítky pro přesnější rozdělení, jak moc je chyba závažná. U těchto nálezů se objevovaly 3 možnosti [71]:

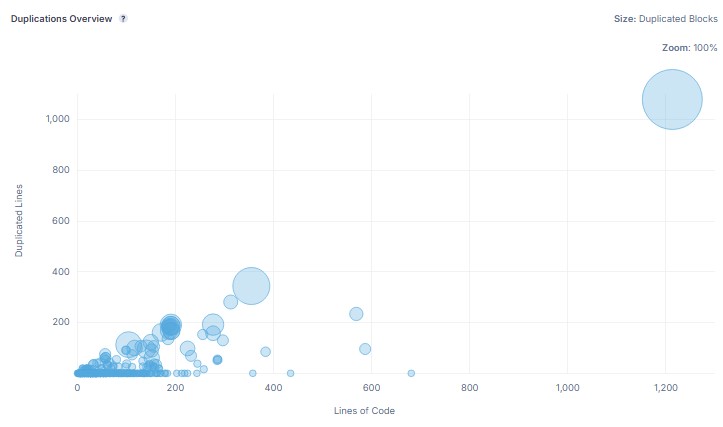
* **Critical** – Kritická chyba, která buď může s určitou pravděpodobností negativně ovlivnit chování aplikace v produkčním prostředí, nebo představuje bezpečnostní riziko.
* **Major** – Závažnější kvalitativní vada kódu, která může výrazně ovlivnit produktivitu vývojáře.
* **Minor** – Méně závažná kvalitativní vada kódu, která má pouze malý dopad na produktivitu vývojáře.

V „Maintainability se nacházelo 695 zranitelností. Z toho 5 kritických, 120 s vysokou závažností, 254 se střední závažností, 291 s nízkou závažností a 25 zranitelností zařazených do „Info“.

V sekcích „Security“ i „Reliability“ se nevyskytovaly žádné nálezy přiřazené do „Info“. Tyto nálezy se objevovaly pouze v „Maintainability“. Všechny nálezy obsahovaly stejný popisek. Jednalo se o úkol, který měl komentář „TODO“. Nástroj tedy vyzíval k jeho dokončení. SonarQube tyto položky přiděluje do „Info“, protože nepředstavují přímé bezpečnostní riziko pro integritu aplikace, ale slouží pouze pro informaci [72].

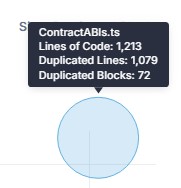
Předposlední metrikou jsou duplikace. Aplikace zobrazuje jak procento duplicitních řádků, tak počet duplicitních řádků, bloků a souborů. Dále poskytuje graf, který zobrazuje dlouhodobý dopad duplicitního kódu na projekt.

Graf dopadu duplicitního kódu na projekt lze vidět na obrázku 69. Velikost každé bubliny reprezentuje množství duplicitních celků a její výška v grafu ukazuje, jak jsou tyto duplicity rozsáhlé (počet řádků). Cílem je mít co nemenší bubliny co nejníže, což značí minimální výskyt opakujícího se kódu.



Obrázek 70 Graf duplicitního kódu v SonarQube

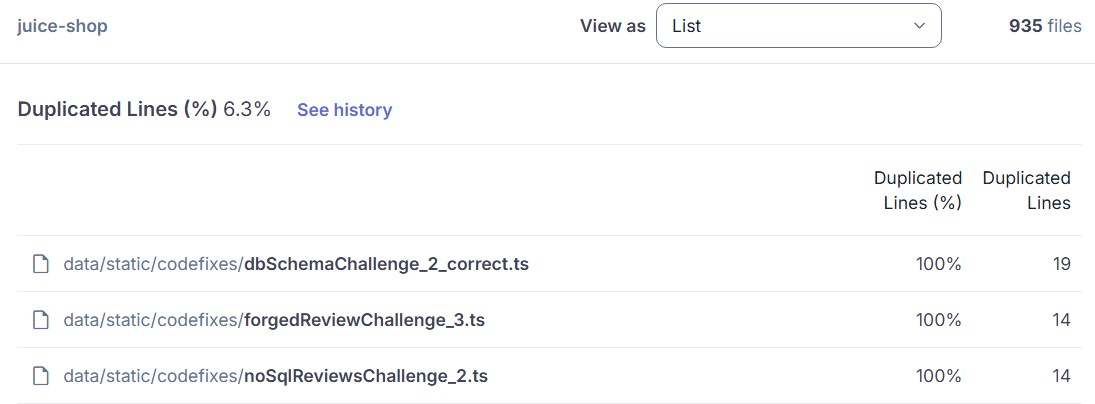
Při umístění kurzoru na jednu z bublinek se vypsalo více informací. Jak lze vidět na obrázku 70, ve výpisu bylo obsažen název konkrétního souboru, ve kterém duplicity byly nalezeny, celkový počet řádků v tomto souboru, počet řádků, které se v projektu vyskytují vícekrát, a počet ucelených opakujících se úseků kódu.



Obrázek 71 Informace o duplicitách v určitém souboru

Nástroj v sekci duplicit obsahuje menu, kde lze vidět jednotlivé rozřazení do „Density“, „Duplicated Lines“, „Duplicated Blocks“ a „Duplicated Files“. Již v tomto menu lze vidět u jednotlivých možností hodnoty.

V možnosti „Density“ bylo vidět jak procento duplicit, které činilo 6,3 %, tak jednotlivé soubory, kde u každého z nich bylo uvedeno procento duplicitních a jejich celkový počet. Zobrazení bylo řešeno pomocí listu. Část tohoto výpisu lze vidět na obrázku 71.



Obrázek 72 Část seznamu souborů, ve kterých se vyskytují duplicity

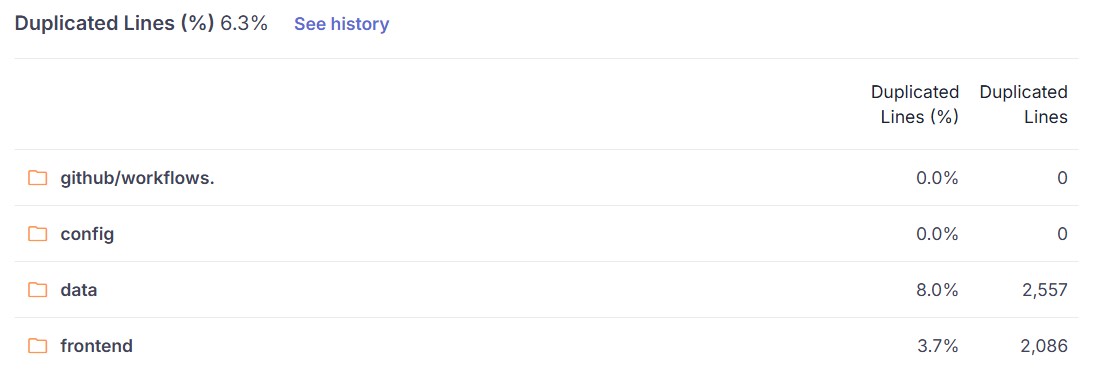
Každý soubor obsahoval i více informací. Po rozkliknutí prvního souboru se zobrazila část kódu, který je 100% duplicitní, což je indikováno šedým pruhem a čísel řádků. Červené podtržení na řádcích 9 a 10 až 13 signalizuje nalezenou chybu. Chyby jsou popsány pod šedými tečkami vedle čísel řádků. Na řádku 9 pod tečkou nástroj informuje, že daný řádek vykazuje nedostatky v čitelnosti záměru, neboť zapsaný kód nedává z logického hlediska smysl. A na řádcích 10 až 13 kód vykazuje nedostatky v jednotnosti kódu, neboť jeho zápis je v rozporu s běžnými programátorskými zvyklostmi. Tento výpis lze vidět na obrázku 72.



Obrázek 73 Výpis kódu s duplicitami

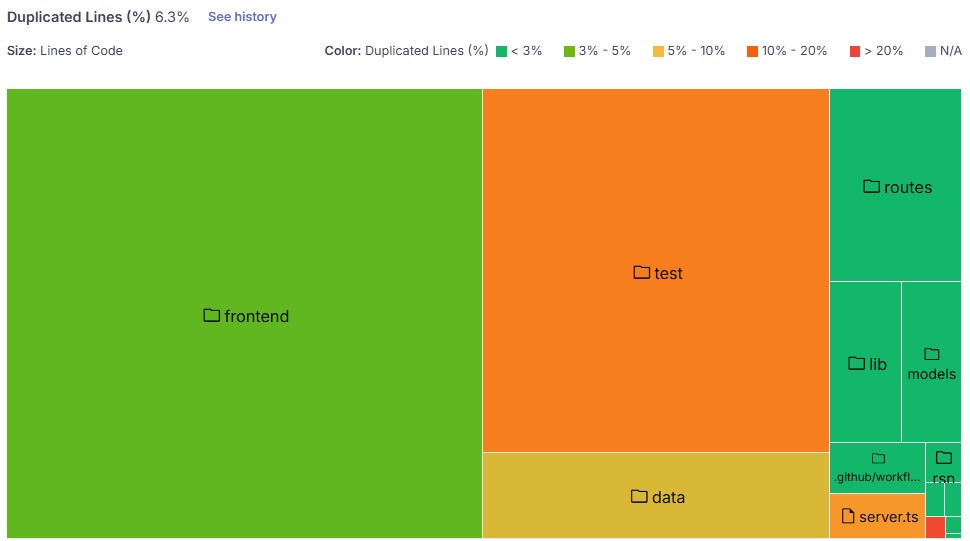
Kromě zobrazení pomocí listu byly na výběr další dvě možnosti zobrazení, a to „Tree“ a „Treemap“.

Pod možností „Tree“ nebyly vypsány jednotlivé soubory, které obsahují duplicity, ale všechny byly v adresářích, viz obrázek 73.



Obrázek 74 Zobrazení duplicit pomocí „Tree“

Poslední možnost „Treemap“ zobrazuje hierarchickou mapu projektu, která vizualizuje distribuci duplicitního kódu. Velikost jednotlivých bloků odpovídá rozsahu kódu (počet řádků) v daných modulech, zatímco barva indikuje míru duplicity. Z grafu je patrné, že největší modul „frontend“ vykazuje přijatelnou míru duplicit (zelená barva), zatímco složka „test“ představuje oblast s vyšším technickým dluhem (oranžová barva), kde by bylo vhodné provést optimalizaci vnitřní struktury pro zvýšení efektivity údržby (viz obrázek 74).



Obrázek 75 „Treemap“ zobrazení duplicit

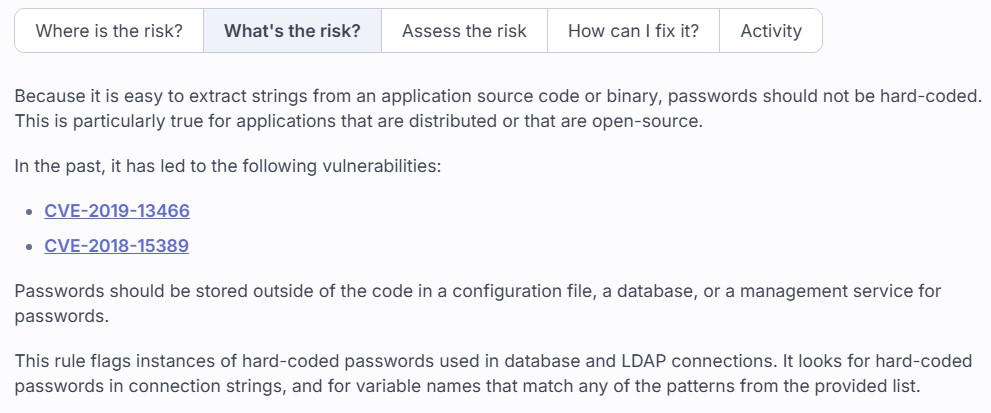
V rámci nástroje jsou poslední sekcí tzv. „Security Hotspots“, tedy metrika, podle které SonarQube identifikuje potenciálně citlivá místa v kódu, jež vyžadují manuální posouzení vývojářem.

Nalezené incidenty byly zařazeny podle priority od nejvyšší po nejnižší. Priorita určuje, jak naléhavě je potřeba kód prověřit. Tato klasifikace vychází z uznávaných standardů, jako jsou OWASP Top 10 nebo CWE Top 25 [73].

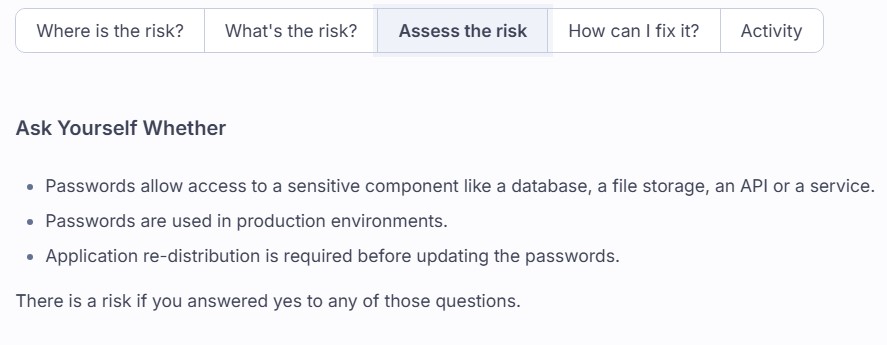
Pro vybranou zranitelnou aplikaci nástroj nalezl do priority „Hight“ 171 zranitelností pod názvem „Authentication“, kde byly převážně zranitelnosti pevně zakódovaného hesla (CWE-259) a 9 „Cross-Site Scripting“. Priorita „Medium“ zahrnuje 10 případů rizika **Denial of Service** (odmítnutí služby), 6 případů **Code Injection** (vkládání kódu) a 11 nálezů souvisejících se **slabým šifrováním**. Tyto chyby jsou sice závažné, ale jejich zneužití často vyžaduje specifické podmínky nebo složitější přípravu útočníka. A jako poslední priorita „Low“ obsahovala 42 nálezů v oblasti šifrování citlivých dat a 2 chyby v zabezpečení konfigurace.

Každá zranitelnost měla tři specializované záložky, které vedou analytika k informovanému rozhodnutí:

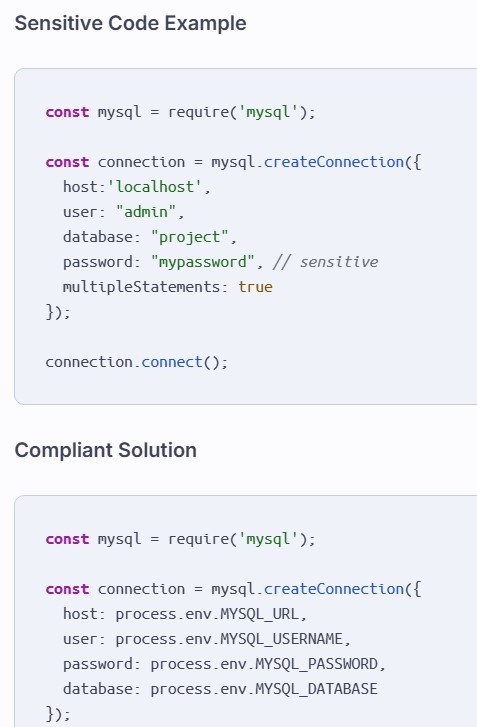
* **„What’s the risk?“** – Karta, která vysvětlovala teoretické pozadí a důvody, proč byl daný úsek kódu označen jako potenciálně nezabezpečený, viz   
  obrázek 75.
* **„Assess the risk“** – Zde byly formulovány kontrolní otázky, které měly pomoct určit, zda byl zvýrazněný kód v daném konkrétním prostředí skutečně zneužitelný, nebo zda se jednalo o bezpečné využití vykazující pouze znaky rizikového vzoru. Tuto kartu lze vidět na obrázku 76.
* **„How can you fix it?“** – Tato karta poskytovala doporučené postupy a příklady bezpečného kódování, které slouží jako návod pro opravu. Dodržení těchto standardů zajišťuje, že nová verze kódu již nebude obsahovat původní slabinu. Část výpisu lze vidět na obrázku 77.



Obrázek 76 Karta „What’s the risk?“ u vybraného hotspotu



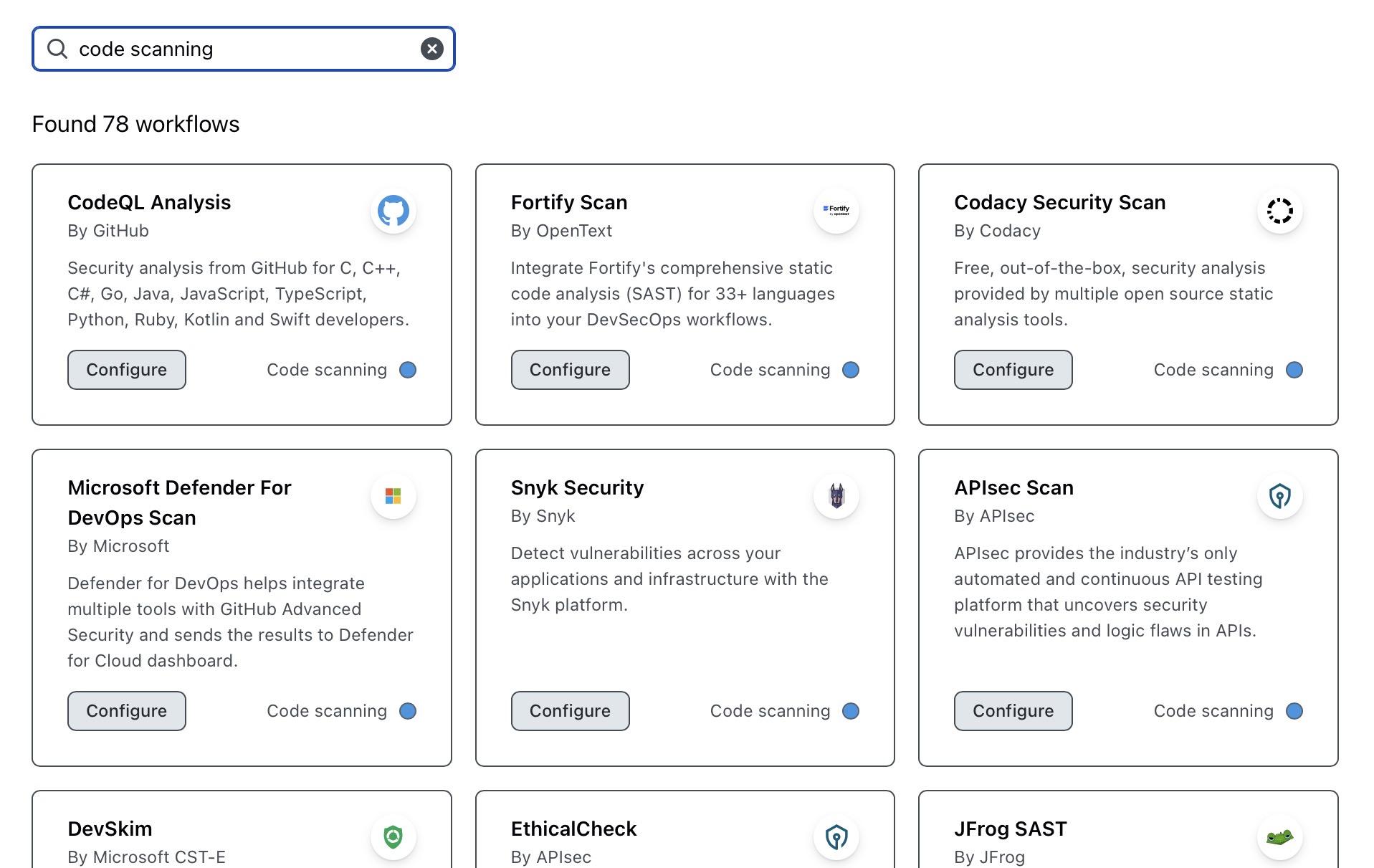
Obrázek 77 Karta „Assess the risk“ u vybraného hotspotu



Obrázek 78 Část výpisu z karty „How can I fix it?“ u vybraného hotpostu

### CodeQL

Dalším nástrojem, který byl otestován, byl nástroj CodeQL. Tento nástroj byl použit přímo ve webové aplikaci GitHub, kde byl fork repozitáře zranitelné aplikace. V sekci „Security“ a kartě „Code scanning“ hned nahoře byl odkaz na přidání nástroje. Po rozkliknutí tohoto odkazu se zobrazilo vyhledávací pole a seznam nástrojů, které lze v GitHub použít, viz obrázek 78.



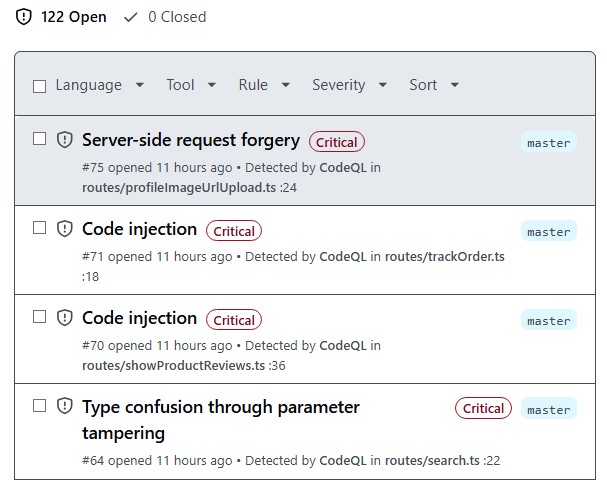
Obrázek 79 Část výpisu nástrojů

Po vybrání nástroje CodeQL se automaticky vygeneroval konfigurační soubor *codeql.yml.* Tento soubor definoval parametry analýz v rámci CI/CD. Před commitem tohoto souboru byly provedeny menší změny. Byl smazán řádek, kde bylo nakonfigurováno „schedule“, že analýza bude probíhat vždy pouze v sobotu v 8:31. Výpis 4 zobrazuje kód tohoto konfiguračního souboru.

Výpis 4 Konfigurační kód pro analýzu CodeQL



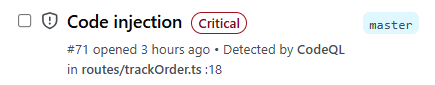
Analýza automaticky proběhla hned po commitu a její výsledky byly zapsány opět v sekci „Security“ a kartě „Code scanning“. I tento nástroj umožňoval filtrování zranitelností. Filtry byly podle programovacího jazyka, nástroje, role, závažnosti anebo od nejnovějšího nálezu a naopak. Část výpisu a filtry lze vidět na obrázku 79.



Obrázek 80 Část výpisu zranitelností z nástroje CodeQL

Analyzátor nalezl 122 zranitelností, z toho 7 kritické závažnosti, 75 vysoké závažnosti, 40 střední závažnosti a zranitelnosti závažnosti nízké nebyly nalezeny žádné.

Jednou z nich byla zranitelnost „Code injection“ na níž byly otestovány ostatní funkce aplikace. Již v náhledu bylo možné vidět, v jakém souboru a na jakém řádku se zranitelnost nachází, viz obrázek 80.



Obrázek 81 Příklad náhledu nálezu nástroje CodeQL

Rozkliknutím dané zranitelnosti se zobrazil výpis kódu se zranitelností a chyba byla zvýrazněna žlutou barvou. Přímo u výpisu aplikace poskytovala i odkaz na zobrazení cesty. Toto zobrazení zobrazuje tok dat, tedy trasu, kterou urazí uživatelský vstup od svého začátku až do místa, kde může způsobit škodu. Část této cesty lze vidět na obrázku 81.



Obrázek 82 Výpis kódu s danou zranitelností nástrojem CodeQL

Hlavní užití nástroje CodeQL je přímo ve Visual Studio Code editoru. Pro analýzu bylo nejdříve třeba připravit si nástroj k použití. Ve VS Code v sekci „Extentions“ byl vyhledán CodeQL a následně nainstalován. Následně pro správné fungování byl nainstalován soubor *vscode-codeql-v1.17.7.vsix* z GitHub odkaz. Následně opět v sekci „Extensions“ v menu pod třemi tečkami byla zvolena volba „Install from VSIX“ a vybrán nainstalovaný soubor.

### Horusec

Prvním nástrojem, který byl otestován pomocí Docker a přes CLI, byl nástroj Horusec. Byl použit pomocí návodu od uživatele GtiHub ZupIT a jeho repozitáře horusec [74]. Nejprve bylo nutné si nástroj instalovat. Instalace proběhla pomocí uvedeného příkazu v repozitáři pro Windows. Výpis 5 zobrazuje tento příkaz.

Výpis 5 Příkaz pro instalaci nástroje Horusec

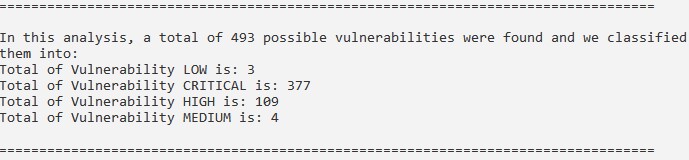


Pro spuštění muselo být nejdříve provedeno přesměrování do složky, kde se zranitelná aplikace nachází a následně pomocí příkazu, který lze vidět ve výpisu 6, spustit analýzu.

Výpis 6 Příkaz pro spuštění analýzy Horusac nástroje



Během pár sekund se všechny nalezené zranitelnosti vypsaly do příkazového řádku. Na spodku výpisu byl vypsán souhrn, kde bylo vypsáno, že bylo nalezeno 493 zranitelností. Z těchto všech nalezených zranitelností bylo 377 kritické závažnosti, 109 vysoké závažnosti, 4 střední a 3 nízké závažnosti. Tento souhrn lze vidět na   
obrázku 82.



Obrázek 83 Souhrn nálezů pomocí nástroje Horusec

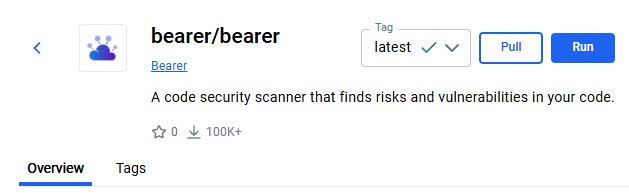
Výše byly jednotlivé nálezy. Každá nalezená zranitelnost obsahovala informace, v jakém jazyce byla naprogramována, její úroveň závažnosti, na jakém řádku se nachází, o kolikátý znak se jedná, jaký nástroj byl použit pro nalezení dané zranitelnosti, jakou míru jistoty nástroj měl, kde se zranitelnost nachází, a také konkrétní řádek kódu. Tento výpis informací pro vybranou zranitelnost lze vidět na obrázku 83.



Obrázek 84 Informace o jedné z nalezených zranitelností nástrojem Horusec

### Bearer

Bearer byl další nástroj, který byl použit v CLI. K analýze byl třeba Docker, ve kterém v sekci „Docker Hub“ byl nalezen nástroj Bearer, a následně pomocí tlačítka „Run“ spuštěn. Část této stránky lze vidět na obrázku 84.



Obrázek 85 Úvodní strana nástroje Bearer

Na téže stránce byl popis nástroje a odkazy na dokumentaci, GitHub a Discord. Také se zde vyskytoval samotný odkaz ke spuštění analýzy z příkazového řádku.

Po přejití na samotný kontejner nástroje se zobrazovaly instrukce k analýze. Jednalo se o příklady, jak analýzu spustit, a příkazy, které lze použít, a další informace. Výpis 7 tyto informace zobrazuje.

Výpis 7 Informace k použití nástroje Bearer



Následně byl zadán příkaz, který lze vidět ve výpisu 8, pro spuštění samotné analýzy, která trvala 4 minuty a 53 sekund.

Výpis 8 Příkaz pro spuštění analýzy nástroje Bearer



Během analýzy byl vidět celý progres pomocí procent. Následně v samotném reportu bylo vypsáno, že nástroj zanalyzoval 1194 souborů v jazyce JavaScript a 6 v jazyce Python. Následně byly vypsány všechny nalezené zranitelnosti. Každá nalezená zranitelnost obsahovala informace o levelu závažnosti, název zranitelnosti, CWE ID, příkaz pro ignorování nálezu, ve kterém souboru se zranitelnost nachází vč. řádku, a v poslední řadě vypsaný samostatný řádek se zranitelností. Příklad výpisu lze vidět ve výpisu 9.

Výpis 9 Jeden z nálezů nástroje Bearer



Na konci výpisu se zobrazoval souhrn nálezů. Sken nalezl dohromady 688 zranitelnosti a z toho 42 kritické závažnosti, 42 vysoké, 68 střední a 536 nízké závažnosti a u každého levelu závažnosti bylo v závorce sepsáno, o jaké CWE se jednalo, viz výpis 10.

Výpis 10 Souhrn nálezů nástroje Bearer



Kromě tohoto reportu nástroj Bearer umožňuje i report uložit do webové stránky. V tomto případě byl použit příkaz, který lze vidět ve výpisu 11.

Výpis 11 Příkaz pro vytvoření HTML souboru a uložení reportů



Na stránce s výpisem se vyskytovaly nejdříve počty zranitelností rozdělené podle levelu závažnosti a následně již byly jednotlivé nálezy.

Seznam nálezů byl seskládán pod sebou od nejkritičtějších zranitelností po nejmíň závažné.

Každá nalezená zranitelnost obsahovala informace od typu závažnosti až po externí zdroje pro hlubší vhled do problematiky. Hned pod názvem zranitelnosti byly podrobnější informace o daném nálezu.

Jednou z těchto informací bylo zmínění pravidla ID, které bylo u každého nálezu rozděleno na několik částí. Tyto části byly rozděleny na programovací jazyk, ve kterém tato zranitelnost byla naprogramována, jaká knihovna byla použita, jakou funkci zranitelnost využívá a popřípadě na další části. Příklad tohoto pravidla lze vidět ve výpisu 12.

Výpis 12 Pravidlo ID



Další důležitou informací obsaženou ve výpisu bylo umístění zranitelnosti a konkrétní řádek byl i vypsán, příklad výpisu řádku lze vidět ve výpisu 13.

Výpis 13 Příklad výpisu řádku se zranitelností nástrojem Bearer



Velmi užitečnou částí tohoto výpisu byla část, jak zranitelnost opravit. Nejprve bylo vypsáno, jak vypadá nebezpečný kód. Příkladem je použití funkce *eval()* pro spuštění kódu přímo se vstupem uživatele, který lze vidět ve výpisu 14.

Výpis 14 Příklad nebezpečného použití funkce vygenerovaný nástrojem Bearer



Následně bylo nástroje doporučeno, co by mělo být použito namísto nebezpečného kódu. Například pro výše uvedenou chybu nástroj vygeneroval bezpečnou variantu, kterou lze vidět vy výpisu 15.

Výpis 15 Bezpečný kód pro danou nalezenou zranitelost



Výpis zranitelnosti také obsahoval informaci, o jaké CWE se jedná, odkazy na externí zdroje pro více informací a další.

## Výsledky a diskuze

Následně tabulky s výsledky a komentář.

# Závěr

Tato práce se zabývala porovnáním vybraných SAST nástrojů, které jsou v dnešní době nevyhnutelnou součástí vývoje softwaru. V rámci analýzy byly nástroje hodnoceny podle různých kritérií, jako je počet podporovaných programovacích jazyků, dostupnost jako open source, možnosti integrace do vývojového procesu   
a podpora automatizovaného testování.

Porovnání ukázalo, že jednotlivé nástroje se liší nejen ve funkcionalitě a rozšířenosti, ale také v dostupnosti pro různé typy projektů. Výběr vhodného SAST nástroje tedy závisí na konkrétních potřebách týmu a projektu – například na preferenci open source řešení, požadavcích na podporované jazyky nebo na možnosti integrace do CI/CD pipeline.

Celkově práce potvrzuje, že využití SAST nástrojů významně přispívá ke zvýšení kvality a bezpečnosti softwaru, umožňuje včas odhalit chyby a zranitelnosti a je důležitým prvkem moderního vývojového procesu. V budoucnu by bylo vhodné zaměřit se na kombinaci SAST s dalšími metodami testování a analýzy kódu, aby byla zajištěna co nejvyšší efektivita a pokrytí kódu.

Seznam použitých zdrojů

1. JACKSON, Gita; KOSINSKI, Matthew a HOLDSWORTH, Jim. *What is the software development life cycle (SDLC)?* Online. IBM. 2025. Dostupné z: <https://www.ibm.com/think/topics/sdlc>. [cit. 2025-08-26].
2. TUTORIALSPOINT. *SDLC - Waterfall Model*. Online. Dostupné z: <https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm>. [cit. 2025-09-14].
3. GEEKSFORGEEKS. *SDLC V-Model - Software Engineering*. Online. 2025-08-11. Dostupné z: <https://www.geeksforgeeks.org/software-engineering/software-engineering-sdlc-v-model>. [cit. 2025-09-14].
4. LAOYAN, Sarah. *What is Agile methodology? (A beginner’s guide)*. Online. 2025. Dostupné z: <https://asana.com/resources/agile-methodology>. [cit. 2025-09-14].
5. PRODUCTPLAN. *Lean Software Development*. Online. Dostupné z: <https://www.productplan.com/glossary/lean-software-development>. [cit. 2025-09-14].
6. SCALER. *SDLC - Iterative Model*. Online. 2023-05-22. Dostupné z: <https://www.scaler.com/topics/software-engineering/iterative-model-in-software-engineering>. [cit. 2025-09-14].
7. HASHEMI-POUR, Cameron. *What is the spiral model and how is it used?* Online. 2025. Dostupné z: <https://www.techtarget.com/searchsoftwarequality/definition/spiral-model>. [cit. 2025-09-14].
8. GEEKSFORGEEKS. *Overview of Big Bang Model*. Online. 2025-07-23. Dostupné z: <https://www.geeksforgeeks.org/software-engineering/overview-of-big-bang-model>. [cit. 2025-09-14].
9. KISSFLOW, Team. *What is Rapid Application Development (RAD)? An Ultimate Guide for 2025*. Online. 2025-08-11. Dostupné z: <https://kissflow.com/application-development/rad/rapid-application-development>. [cit. 2025-09-14].
10. *What is version control?* Online. Atlassian. Dostupné z: <https://www.atlassian.com/git/tutorials/what-is-version-control>. [cit. 2025-08-27].
11. ATLASSIAN. *What is Git?* Online. Dostupné z: <https://www.atlassian.com/git/tutorials/what-is-git>. [cit. 2025-10-02].
12. VALKOVIČ, Patrik. *Lekce 1 - Git - Historie a principy*. Online. Dostupné z: <https://www.itnetwork.cz/programovani/git/git-tutorial-historie-a-principy>. [cit. 2025-10-07].
13. MICROSOFT. *Co je DevOps?* Online. Dostupné z: <https://azure.microsoft.com/cs-cz/resources/cloud-computing-dictionary/what-is-devops>. [cit. 2025-10-18].
14. RED HAT. *What is CI/CD?* Online. 2025-06-10. Dostupné z: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>. [cit. 2025-10-18].
15. THIRMAL, Adhiran. *What is Secure Development?* Online. 2023. Dostupné z: <https://www.securitycompass.com/blog/what-is-secure-development>. [cit. 2025-10-18].
16. KONDAKOVA, Radostina. *Secure Software Development: Overview and practical examples*. [online]. 2025 [cit. 2025-11-08]. Dostupné z: <https://owasp.org/www-chapter-sofia/assets/presentations/202503%20-%20Secure%20Software%20Development:%20Overview%20and%20practical%20examples%20by%20Radostina%20Kondakova.pdf>
17. TAL, Liran. *Secure Software Development Lifecycle (SSDLC)*. Online. Snyk Limited. Dostupné z: <https://snyk.io/articles/secure-sdlc>. [cit. 2025-11-09].
18. DRAKE, Victoria. *Threat Modeling*. Online. OWASP. Dostupné z: <https://owasp.org/www-community/Threat_Modeling>. [cit. 2025-11-09].
19. BLACK DUCK SOFTWARE. *Software Composition Analysis*. Online. Black Duck. Dostupné z: <https://www.blackduck.com/glossary/what-is-software-composition-analysis.html>. [cit. 2025-11-08].
20. GOODWIN, Michale a KHAN, Tasmiha. *What is a software bill of materials (SBOM)?* Online. IBM. Dostupné z: <https://www.ibm.com/think/topics/sbom>. [cit. 2025-11-08].
21. KEYFACTOR. *Introducing the Cryptographic Bill of Materials (CBOM): A Foundation for Modern Cryptographic Management*. Online. Keyfactor. 2025. Dostupné z: <https://www.keyfactor.com/blog/introducing-the-cryptographic-bill-of-materials-cbom-a-foundation-for-modern-cryptographic-management>. [cit. 2025-11-08].
22. GEEKSFORGEEKS. *What is Secure Software Development Life Cycle (SSDLC )?* Online. GeeksforGeeks. 2025-08-09. Dostupné z: <https://www.geeksforgeeks.org/ethical-hacking/what-is-secure-software-development-life-cycle-ssdlc>. [cit. 2025-11-09].
23. FLEXERA. *Software vulnerability*. Online. Dostupné z: <https://www.flexera.com/resources/glossary/software-vulnerability>. [cit. 2025-10-18].
24. RED HAT. *What is a CVE?* Online. 2024. Dostupné z: <https://www.redhat.com/en/topics/security/what-is-cve>. [cit. 2025-10-24].
25. FORTINET. *Common Vulnerabilities and Exposures (CVE): Everything You Need to Know*. Online. Dostupné z: <https://www.fortinet.com/resources/cyberglossary/cve>. [cit. 2025-10-24].
26. KHAN, Tasmiha a GOODWIN, Michael. *What is CVE (Common Vulnerabilities and Exposures)?* Online. Dostupné z: <https://www.ibm.com/think/topics/cve>. [cit. 2025-10-24].
27. MICROSOFT. *Microsoft Outlook Elevation of Privilege Vulnerability*. Online. 2023, 2023-03-21. Dostupné z: <https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-23397?utm_source=chatgpt.com>. [cit. 2025-10-26].
28. WHATSAPP. *WhatsApp Security Advisories*. Online. 2025. Dostupné z: <https://www.whatsapp.com/security/advisories/2025>. [cit. 2025-10-26].
29. CLOUDFLARE. *What was the WannaCry ransomware attack?* Online. Dostupné z: <https://www.cloudflare.com/learning/security/ransomware/wannacry-ransomware>. [cit. 2025-10-26].
30. MICROSOFT. *Windows SMB Remote Code Execution Vulnerability*. Online. Microsoft. 2017. Dostupné z: <https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2017-0144>. [cit. 2025-11-09].
31. BUGCROWD. *Common Weakness Enumeration (CWE)*. Online. Bugcrowd. Dostupné z: <https://www.bugcrowd.com/glossary/common-weakness-enumeration-cwe>. [cit. 2025-11-09].
32. MITRE. *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*. Online. 2025-09-09. Dostupné z: <https://cwe.mitre.org/data/definitions/79.html>. [cit. 2025-11-09].
33. KRISTENS. *Cross Site Scripting (XSS)*. Online. OWASP. Dostupné z: <https://owasp.org/www-community/attacks/xss>. [cit. 2025-11-09].
34. GOODMAN, Courtney. *What is the Common Vulnerability Scoring System (CVSS)?* Online. Balbix. 2024-10-25. Dostupné z: <https://www.balbix.com/insights/understanding-cvss-scores>. [cit. 2025-11-09].
35. BLACK DUCK. *Secure Code Review*. Online. Dostupné z: <https://www.blackduck.com/glossary/what-is-code-review.html>. [cit. 2025-08-27].
36. FOSTER, Stuart. Online. 2023. Dostupné z: <https://www.perforce.com/blog/sca/what-static-analysis>. [cit. 2025-10-28].
37. DEWHURST, Ryan. *Static Code Analysis*. Online. OWASP. Dostupné z: <https://owasp.org/www-community/controls/Static_Code_Analysis>. [cit. 2025-11-09].
38. JIT. *Static Application Security Testing (SAST): What You Need to Know*. Online. Jit. 2025. Dostupné z: <https://www.jit.io/resources/appsec-tools/static-application-security-testing-sast-what-you-need-to-know>. [cit. 2025-11-09].
39. GEEKSFORGEEKS. *Introduction of Lexical Analysis*. Online. GeeksforGeeks. 2025-08-26. Dostupné z: <https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis>. [cit. 2025-11-10].
40. TUTORIALSPOINT. *Natural Language Processing - Syntactic Analysis*. Online. Tutorialspoint. Dostupné z: <https://www.tutorialspoint.com/natural_language_processing/natural_language_processing_syntactic_analysis.htm>. [cit. 2025-11-08].
41. GEEKSFORGEEKS. *Understanding Semantic Analysis - NLP*. Online. GeeksforGeeks. 2021-11-28. Dostupné z: <https://www.geeksforgeeks.org/nlp/understanding-semantic-analysis-nlp>. [cit. 2025-11-08].
42. LAZAROV, Willi, 2023. *Bezpečnostní analýza zdrojového kódu*. PDF. 1-68. [cit. 2025-11-10].
43. GITLAB DOCS. *Static Application Security Testing (SAST)*. Online. Jit. 2025. Dostupné z: <https://docs.gitlab.com/user/application_security/sast>. [cit. 2025-08-27].
44. GITHUB. *About code scanning with CodeQL*. Online. GitHub Docs. Dostupné z: <https://docs.github.com/en/code-security/code-scanning/introduction-to-code-scanning/about-code-scanning-with-codeql>. [cit. 2025-11-10].
45. GITHUB. *CodeQL*. Online. GitHub. Dostupné z: [https://codeql.github.com](https://codeql.github.com/). [cit. 2025-11-10].
46. GITHUB. *CodeQL CLI CSV output*. Online. GitHub Docs. Dostupné z: <https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/csv-output?utm_source=chatgpt.com>. [cit. 2025-11-10].
47. GITLUB DOCS. *Static application security testing (SAST)*. Online. GitLub Docs. Dostupné z: <https://docs.gitlab.com/user/application_security/sast>. [cit. 2025-11-10].
48. GITLUB DOCS. *Web IDE*. Online. GitLub Docs. Dostupné z: <https://docs.gitlab.com/user/project/web_ide/?utm_source=chatgpt.com>. [cit. 2025-11-10].
49. SONAR. *Detect vulnerabilities other SAST tools miss*. Online. Sonar. Dostupné z: https://www.sonarsource.com/solutions/security/sast. [cit. 2025-11-10].
50. SONAR. *Developer tools for every language*. Online. Sonar. Dostupné z: https://www.sonarsource.com/knowledge/languages. [cit. 2025-11-10].
51. SONAR. *Detect vulnerabilities other SAST tools miss*. Online. Sonar docs. Dostupné z: https://docs.sonarsource.com/sonarqube-server/2025.4/design-and-architecture/overview. [cit. 2025-11-10].
52. SNYK LIMITED. *Detect vulnerabilities other SAST tools miss*. Online. Snyk. Dostupné z: https://snyk.io/product/snyk-code. [cit. 2025-11-10].
53. SNYK LIMITED. *Supported languages, package managers, and frameworks*. Online. Snyk User Docs. 2025-05. Dostupné z: https://docs.snyk.io/supported-languages/supported-languages-package-managers-and-frameworks. [cit. 2025-11-10].
54. SNYK LIMITED. *Supported IaC languages, cloud providers, and cloud resources*. Online. Snyk User Docs. 2025-02. Dostupné z: https://docs.snyk.io/scan-with-snyk/snyk-iac/supported-iac-languages-cloud-providers-and-cloud-resources?utm\_source=chatgpt.com. [cit. 2025-11-10].
55. SNYK LIMITED. *View Snyk Code CLI results*. Online. Snyk User Docs. 2025-02. Dostupné z: https://docs.snyk.io/developer-tools/snyk-cli/scan-and-maintain-projects-using-the-cli/snyk-cli-for-snyk-code/view-snyk-code-cli-results. [cit. 2025-11-10].
56. CYCODE. *Redefining what code security ‍can do for you*. Online. Bearer. Dostupné z: https://www.bearer.com. [cit. 2025-11-10].
57. CYCODE. *Supported languages*. Online. Bearer. Dostupné z: https://docs.bearer.com/reference/supported-languages/?utm\_source=chatgpt.com. [cit. 2025-11-10].
58. CYCODE. *Set up CI/CD for Bearer CLI*. Online. Bearer. Dostupné z: https://docs.bearer.com/guides/ci-setup. [cit. 2025-11-10].
59. AIKDO. *Aikido Docs Overview*. Online. Bearer. 2025-11-05. Dostupné z: https://help.aikido.dev. [cit. 2025-11-10].
60. AIKDO SECURITY BV. *State-of-the-Art SAST, Built for Developers*. Online. Bearer. 2025-11-05. Dostupné z: https://www.aikido.dev/scanners/static-code-analysis-sast. [cit. 2025-11-10].
61. ZUP IT. *Overview*. Online. Horusec. 2025-11-05. Dostupné z: https://docs.horusec.io/docs/web/overview. [cit. 2025-11-10].
62. MEDIUM. *Horusec in Action: Detecting Vulnerabilities Early with Static Application Security Testing*. Online. Horusec. 2024. Dostupné z: https://medium.com/%40maheshwar.ramkrushna/horusec-is-an-open-source-security-vulnerability-scanner-designed-to-identify-vulnerabilities-in-810ca236a3d1. [cit. 2025-11-10].
63. IN-COM. *Symbolic Execution in Static Code Analysis: A Game-Changer for Bug Detection*. Online. 2025. Dostupné z: https://www.in-com.com/blog/symbolic-execution-in-static-code-analysis-a-game-changer-for-bug-detection. [cit. 2025-11-12].
64. GEEKSFORGEEKS. *Data flow analysis in Compiler*. Online. 2025-07-11. Dostupné z: https://www.geeksforgeeks.org/compiler-design/data-flow-analysis-compiler. [cit. 2025-11-12].
65. TUTORIALSPOINT. *Compiler Design - Control Flow Graph*. Online. Dostupné z: https://www.tutorialspoint.com/compiler\_design/compiler\_design\_control\_flow\_graph.htm. [cit. 2025-11-12].
66. OWASP. *OWASP Juice Shop*. Online. OWASP. Dostupné z: https://owasp.org/www-project-juice-shop/. [cit. 2026-01-02].
67. TRIGGER.DEV. *JSON sucks. But we're making it better*. Online. JSON Hero. Dostupné z: https://jsonhero.io. [cit. 2026-01-02].
68. *Package-lock.json* [@bkimminich]. Online. 2019, 2025-08. Dostupné z: GitHub, https://github.com/juice-shop/juice-shop-ctf/blob/master/package-lock.json. [cit. 2026-01-02].
69. SONARSOURCE SÀRL. *Rules*. Online. Sonar docs. 2025, 2025. Dostupné z: <https://docs.sonarsource.com/sonarqube-cloud/digging-deeper/rules>. [cit. 2026-01-13].
70. SONARSOURCE. *What is a code smell?* Online. SonarSource. Dostupné z: https://www.sonarsource.com/resources/library/code-smells/. [cit. 2026-01-11].
71. SONARSOURCE. *Issues*. Online. SonarSource. 2025-08. Dostupné z: https://docs.sonarsource.com/sonarqube-server/9.9/user-guide/issues. [cit. 2026-01-11].
72. SONARSOURCE SÀRL. *Software qualities*. Online. Sonar docs. 2025. Dostupné z: [https://docs.sonarsource.com/sonarqube-cloud/digging-deeper/software-qualities/?\_gl=1\*1ltscgu\*\_gcl\_au\*MTY0MDI5OTk3LjE3NjUzOTA4NzY.\*\_ga\*OTY0MDU1NDk0LjE3NjUzOTA4Njk.\*\_ga\_9JZ0GZ5TC6\*czE3NjgwMzYwNTkkbzMkZzEkdDE3NjgwMzYwNjMkajU2JGwwJGgw](https://docs.sonarsource.com/sonarqube-cloud/digging-deeper/software-qualities/?_gl=1*1ltscgu*_gcl_au*MTY0MDI5OTk3LjE3NjUzOTA4NzY.*_ga*OTY0MDU1NDk0LjE3NjUzOTA4Njk.*_ga_9JZ0GZ5TC6*czE3NjgwMzYwNTkkbzMkZzEkdDE3NjgwMzYwNjMkajU2JGwwJGgw). [cit. 2026-01-13].
73. SONARSOURCE SÀRL. *Security Hotspots*. Online. Dostupné z: <https://docs.sonarsource.com/sonarqube-server/10.6/user-guide/security-hotspots>. [cit. 2026-01-20].
74. PAWŁOWSKI, Łukasz a DURU, Bruno. *Horusec*. Online. Dostupné z: <https://github.com/ZupIT/horusec>. [cit. 2026-01-21].

|  |  |  |
| --- | --- | --- |
|  |  |  |

Seznam obrázků

[Obrázek 1 Proces SDLC 9](#_Toc221174106)

[Obrázek 2 Vodopádový model 10](#_Toc221174107)

[Obrázek 3 V-model 11](#_Toc221174108)

[Obrázek 4 Agilní model 12](#_Toc221174109)

[Obrázek 5 Lean model 12](#_Toc221174110)

[Obrázek 6 Iterativní model 13](#_Toc221174111)

[Obrázek 7 Spirálový model 14](#_Toc221174112)

[Obrázek 8 Big Bang model 14](#_Toc221174113)

[Obrázek 9 – Rad model 15](#_Toc221174114)

[Obrázek 10 Proces Gitu 19](#_Toc221174115)

[Obrázek 11 Proces DevOps 21](#_Toc221174116)

[Obrázek 12 Proces CI/CD 21](#_Toc221174117)

[Obrázek 13 Proces SSDLC 26](#_Toc221174118)

[Obrázek 14 – Průběh XSS útoku 30](#_Toc221174119)

[Obrázek 15 Proces bezpečné analýzy zdrojového kódu 32](#_Toc221174120)

[Obrázek 16 Proces lexikální analýzy 34](#_Toc221174121)

[Obrázek 17 Proces syntaktické analýzy 34](#_Toc221174122)

[Obrázek 18 Příklad abstraktního syntaktického stromu 35](#_Toc221174123)

[Obrázek 19 Vizualizace abstraktního syntaktického stromu 35](#_Toc221174124)

[Obrázek 20 Graf toku řízení – podmíněné větvení 37](#_Toc221174125)

[Obrázek 21 Graf toku řízení – cyklus 37](#_Toc221174126)

[Obrázek 22 Graf toku řízení – cyklus se zpětnou hranou do A 38](#_Toc221174127)

[Obrázek 23 Úspěšné provedení analýzy nástroje GitLab 43](#_Toc221174128)

[Obrázek 24 Transformovaná výsledná data pomocí aplikace JSON Hero 45](#_Toc221174129)

[Obrázek 25 Nabídka přidání projektu na platformě Snyk 46](#_Toc221174130)

[Obrázek 26 Počet zranitelností jednotlivých stupňů Snyk analýzy 47](#_Toc221174131)

[Obrázek 27 Základní informace projektu a analýzy na platformě Snyk 47](#_Toc221174132)

[Obrázek 28 Filtr zranitelností podle skóre závažnosti v aplikaci Snyk 48](#_Toc221174133)

[Obrázek 29 Ukázka stručného výsledku zranitelnosti z analýzy Snyk 48](#_Toc221174134)

[Obrázek 30 Zvýrazněný řádek se zranitelností v aplikaci GitHub 49](#_Toc221174135)

[Obrázek 31 Tok dat citlivého údaje z analýzy Snyk 50](#_Toc221174136)

[Obrázek 32 Kód se zvýrazněným kritickým řádkem 50](#_Toc221174137)

[Obrázek 33 Teoretický výklad možné opravy chyby 51](#_Toc221174138)

[Obrázek 34 Ukázka kódu s opravami 51](#_Toc221174139)

[Obrázek 35 Formulář pro ignoraci identifikované zranitelnosti 53](#_Toc221174140)

[Obrázek 36 Ukázka zranitelnosti se střední závažností 53](#_Toc221174141)

[Obrázek 37 Ukázka zranitelnosti nízkého stupně 54](#_Toc221174142)

[Obrázek 38 Výpis repozitářů v aplikaci Aikido 55](#_Toc221174143)

[Obrázek 39 Nastavení pro přidání repozitáře z GitHubu do aplikace Aikido 55](#_Toc221174144)

[Obrázek 40 Výběr repozitářů, které mohou být skenovány aplikací Aikido 56](#_Toc221174145)

[Obrázek 41 Repozitáře a jejich provedená analýza pomocí nástroje Aikido 56](#_Toc221174146)

[Obrázek 42 Část výpisu reportů analýzy Aikido 57](#_Toc221174147)

[Obrázek 43 Zranitelnost kritické závažnosti objevená analyzátorem Aikido 58](#_Toc221174148)

[Obrázek 44 Sekce „Education“ v analyzátoru Aikido 59](#_Toc221174149)

[Obrázek 45 Sekce „Subissues“ v nalezené zranitelnosti nástroje Aikido 60](#_Toc221174150)

[Obrázek 46 Ukázka kódu detekované zranitelnosti NoSQL injection 61](#_Toc221174151)

[Obrázek 47 AI souhrn ke zranitelnosti v aplikaci Aikido 61](#_Toc221174152)

[Obrázek 48 Grafické znázornění toku dat zranitelnosti v aplikaci Aikido 62](#_Toc221174153)

[Obrázek 49 Ignorace nalezené zranitelnosti v nástroji Aikido 62](#_Toc221174154)

[Obrázek 50 Formulář pro zpětnou vazbu aplikace Aikido 63](#_Toc221174155)

[Obrázek 51 Funkce „AutoFix preview“ nástroje Aikido 64](#_Toc221174156)

[Obrázek 52 Akce pro každou nalezenou zranitelnost v nástroji Aikido 64](#_Toc221174157)

[Obrázek 53 První formulář funkce pro skrytí zranitelnosti na určitou dobu 65](#_Toc221174158)

[Obrázek 54 Druhý formulář funkce pro skrytí zranitelnosti na určitou dobu 66](#_Toc221174159)

[Obrázek 55 Formulář pro změnu úrovně závažnosti nalezené zranitelnosti 67](#_Toc221174160)

[Obrázek 56 Graf toku dat zranitelnosti s vysokou závažností 68](#_Toc221174161)

[Obrázek 57 Část kódu obsahující zranitelnost vysoké závažnosti 68](#_Toc221174162)

[Obrázek 58 Způsob, jakým lze opravit danou zranitelnost 69](#_Toc221174163)

[Obrázek 59 Výběr definice nového kódu v SonarQube 70](#_Toc221174164)

[Obrázek 60 Náhled zranitelností a ostatní metriky v SonarQube 71](#_Toc221174165)

[Obrázek 61 Náhled jedné z nalezených zranitelností v SonarQube 72](#_Toc221174166)

[Obrázek 62 Menu pro změnu statusu zranitelnosti v SonarQube 73](#_Toc221174167)

[Obrázek 63 Kritická zranitelnost nalezená nástrojem SonarQube 74](#_Toc221174168)

[Obrázek 64 Vysvětlení, proč je daná zranitelnost problém 75](#_Toc221174169)

[Obrázek 65 Příklad nevyhovujícího kódu 75](#_Toc221174170)

[Obrázek 66 Řešení pro příklad nevyhovujícího kódu 76](#_Toc221174171)

[Obrázek 67 Sekce „Activity“ dané zranitelnosti v SonarQube 76](#_Toc221174172)

[Obrázek 68 Nabízené odkazy pro zjištění více informací k dané zranitelnosti 77](#_Toc221174173)

[Obrázek 69 Graf duplicitního kódu v SonarQube 79](#_Toc221174174)

[Obrázek 70 Informace o duplicitách v určitém souboru 80](#_Toc221174175)

[Obrázek 71 Část seznamu souborů, ve kterých se vyskytují duplicity 80](#_Toc221174176)

[Obrázek 72 Výpis kódu s duplicitami 81](#_Toc221174177)

[Obrázek 73 Zobrazení duplicit pomocí „Tree“ 81](#_Toc221174178)

[Obrázek 74 „Treemap“ zobrazení duplicit 82](#_Toc221174179)

[Obrázek 75 Karta „What’s the risk?“ u vybraného hotspotu 83](#_Toc221174180)

[Obrázek 76 Karta „Assess the risk“ u vybraného hotspotu 84](#_Toc221174181)

[Obrázek 77 Část výpisu z karty „How can I fix it?“ u vybraného hotpostu 84](#_Toc221174182)

[Obrázek 78 Část výpisu nástrojů 85](#_Toc221174183)

[Obrázek 79 Část výpisu zranitelností z nástroje CodeQL 87](#_Toc221174184)

[Obrázek 80 Příklad náhledu nálezu nástroje CodeQL 87](#_Toc221174185)

[Obrázek 81 Výpis kódu s danou zranitelností nástrojem CodeQL 88](#_Toc221174186)

[Obrázek 82 Souhrn nálezů pomocí nástroje Horusec 89](#_Toc221174187)

[Obrázek 83 Informace o jedné z nalezených zranitelností nástrojem Horusec 90](#_Toc221174188)

[Obrázek 84 Úvodní strana nástroje Bearer 90](#_Toc221174189)

Seznam tabulek

[Tabulka 1 Úrovně závažnosti podle CVSS 31](#_Toc218968656)

[Tabulka 2 Výstup lexikální analýzy 33](#_Toc218968657)

[Tabulka 3 Srovnání SAST nástrojů 40](#_Toc218968658)

1. Token je sekvence znaků, kterou lze v gramatice programovacího jazyka považovat za jednu jednotku. [↑](#footnote-ref-1)
2. V závislosti na programovacím jazyce se tabulka může ličit. [↑](#footnote-ref-2)
3. CodeQL je dostupné jako rozšíření pro Visual Studio Code a v tomto rozšíření je dostupné GUI, jinak je CodeQL pouze CLI nástroj. [↑](#footnote-ref-3)
4. CI/CD s CodeQL nelze používat na privátních nebo proprietárních projektech, pro běžné softwarové procesy, pro kód, jehož dotazy nejdou OSI-open source. [↑](#footnote-ref-4)
5. GUI je dostupné, ale pouze pro placenou verzi. [↑](#footnote-ref-5)
6. Pro bezplatnou verzi pouze je 6 podporovaných jazyků. [↑](#footnote-ref-6)
7. Vážná bezpečnostní zranitelnost webových aplikací, která útočníkům umožňuje vkládat škodlivé skripty do důvěryhodných webových stránek. [↑](#footnote-ref-7)