

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 mé 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 my 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 30. října 2025 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 30. října 2025

podpis autora

Obsah

[Úvod 5](#_Toc2130715949)

[1 Teoretická část 6](#_Toc1086292808)

[1.1 Životní cyklus vývoje software 7](#_Toc776636142)

[1.1.1 Verzování vývoje 8](#_Toc1319577843)

[1.1.2 Proces DevOps 8](#_Toc1055380097)

[1.1.3 Bezpečný vývoj 9](#_Toc157953512)

[1.2 Softwarové zranitelnosti 10](#_Toc259920018)

[1.2.1 Známe zranitelnosti (CVE) 10](#_Toc1655416579)

[1.2.2 Kategorizace zranitelností (CWE) 10](#_Toc1100154409)

[1.2.3 Hodnocení zranitelností (CVSS) 11](#_Toc2043311482)

[1.3 Bezpečnostní analýza zdrojového kódu 11](#_Toc584401387)

[1.3.1 Statická analýza 11](#_Toc1491016337)

[1.3.2 SAST 11](#_Toc832150965)

[1.3.3 Srovnání vybraných nástrojů 12](#_Toc569457689)

[1.4 Závěr 12](#_Toc888773035)

[1.5 Seznam použitých zdrojů 13](#_Toc1027055236)

[1.6 Seznam použitých symbolů a zkratek 13](#_Toc1420254010)

[1.7 Seznamy použitých obrázků a tabulek 13](#_Toc2013592283)

[1.8 Seznam příloh 13](#_Toc599572802)

[Závěr 13](#_Toc191610483)

[Seznam použitých zdrojů 14](#_Toc1403892123)

[Seznam použitých symbolů a zkratek 17](#_Toc659856044)

[Seznam obrázků 18](#_Toc853737939)

[Seznam tabulek 19](#_Toc719251975)

[Seznam příloh 20](#_Toc143617415)

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, CWE a CVSS. 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

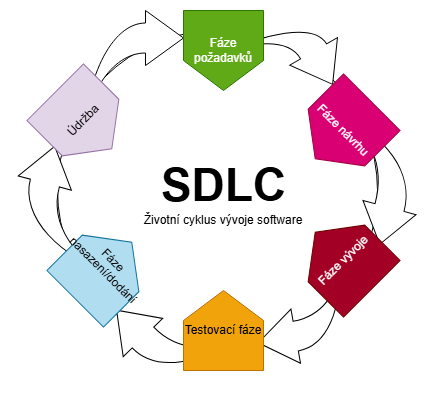
Teoretická část se zaměřuje na bezpečný vývoj software prevenci jeho zranitelností. Úvodem se zaměřuje na životní cyklus vývoje, verzování, DevOps a principy bezpečného kódu. Dále se věnuje softwarovým zranitelnostem, kategorizaci a hodnocení. Závěrem se zaměřuje na bezpečnostní analýzu zdrojového kódu, 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.

Proces SDLC se obvykle dělí do několika fází:

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

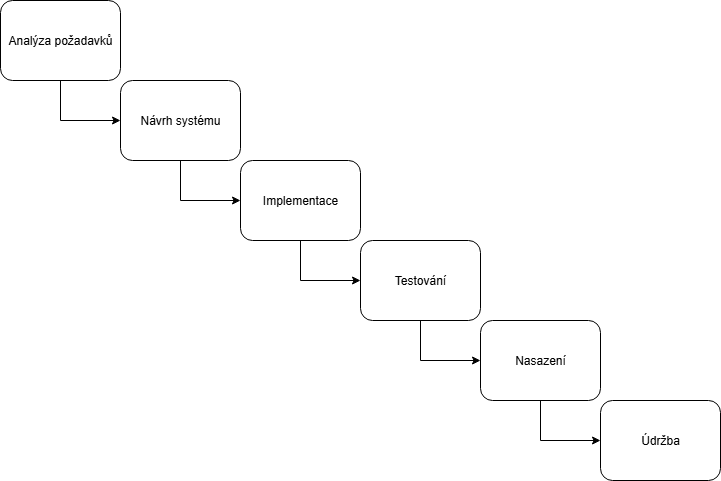


Obrázek 1 – Životní cyklus vývoje software

SDLC má i mnoho modelů:

1. Vodopád

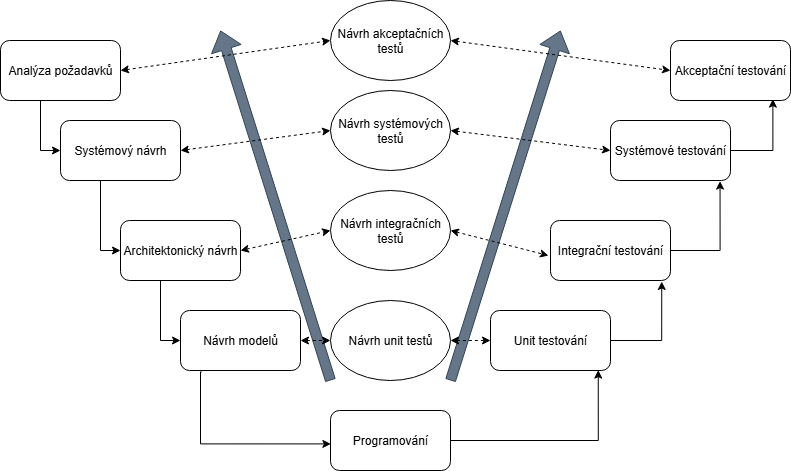
Tento model byl prvním procesním modelem. Označuje se jako lineárně-sekvenční model životního cyklu. Každá fáze musí být dokončena před zahájením další fáze. V modelu Vodopád se fáze nepřekrývají.



Obrázek 2 – Vodopádový model

2. 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.

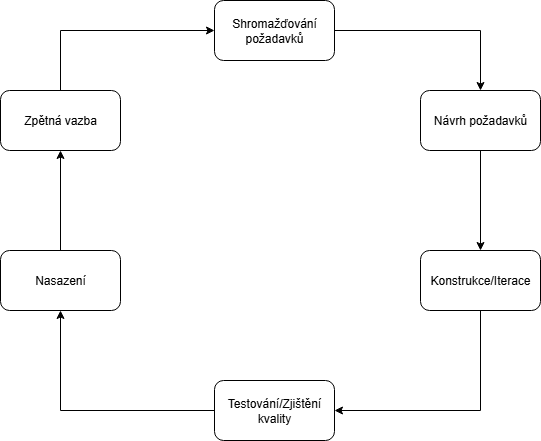


Obrázek 3 – V-model

3. 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.

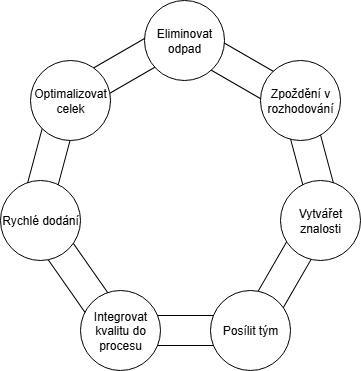
Mezi nejběžnější metodologie patří Kanban a Scrum.



Obrázek 4 – Agilní model

4. Lean

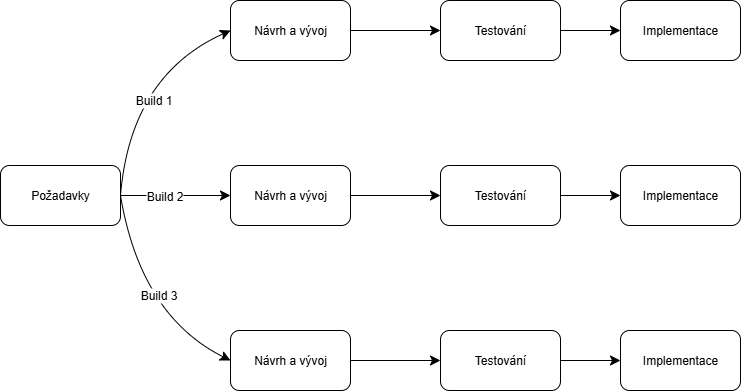
Lean model je založený na optimalizaci času a zdrojů vývoje, eliminaci plýtvání   
a v konečném důsledku dodávání pouze toho, co produkt potřebuje.



Obrázek 5 – Lean model

5. Iterativní model

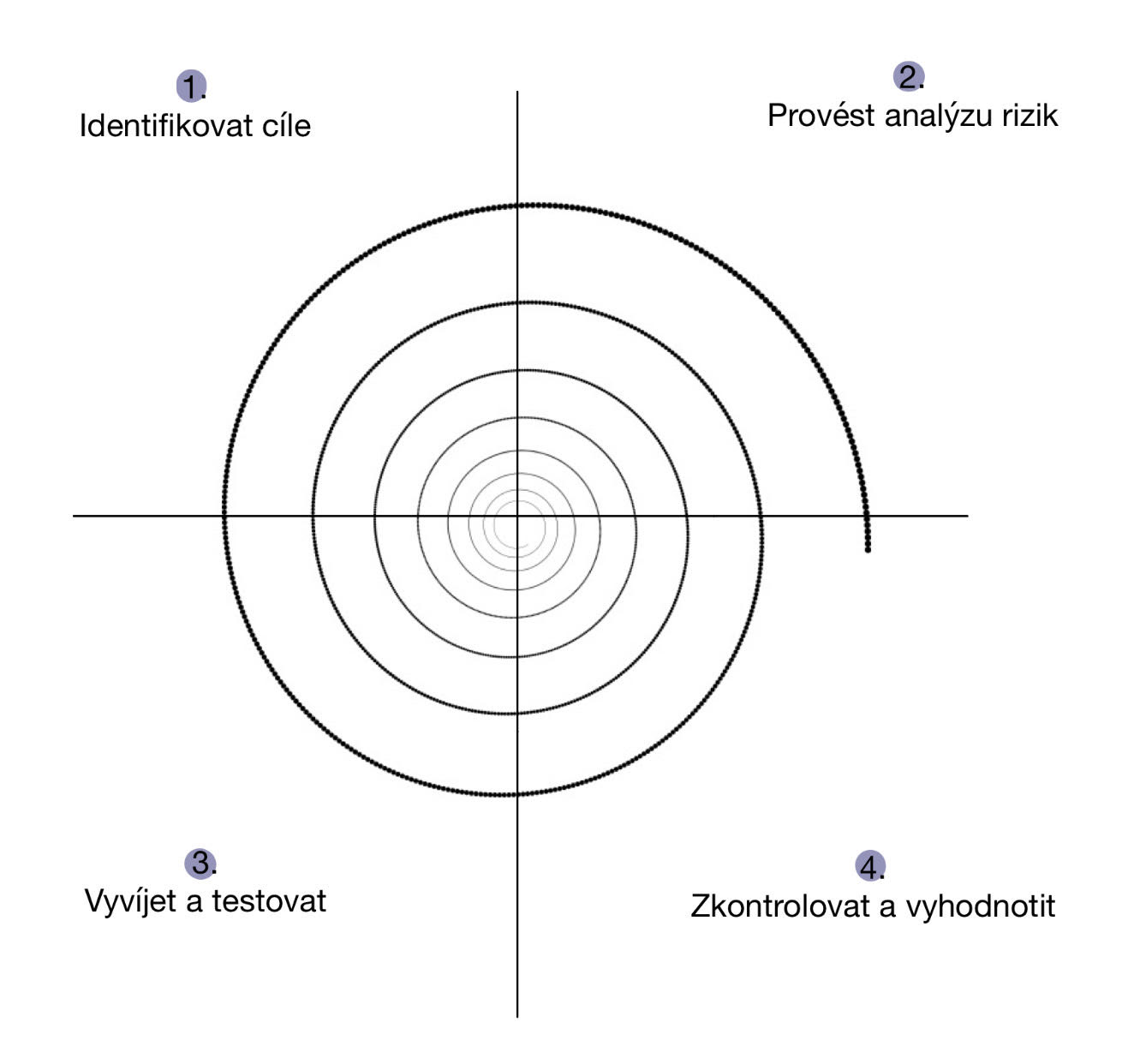
Model, ve kterém se počáteční vývojové práce provádějí na základě dobře stanovených základních požadavků a k tomuto základnímu software se prostřednictvím iterací přidávají následná vylepšení, dokud není vytvořen finální systém.



Obrázek 6 – Iterativní model

6. Spirálový model

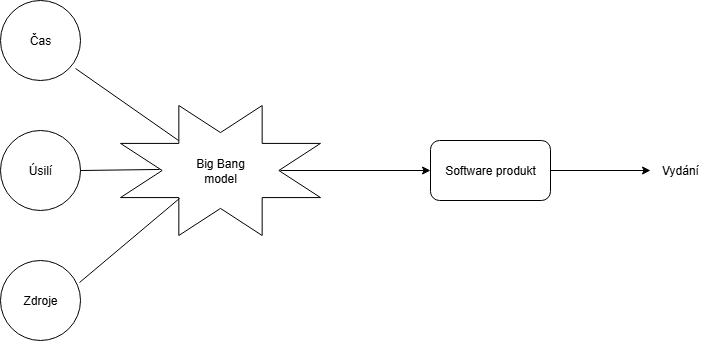
Spirálový model je kombinace iterativního modelu a vodopádového modelu. Používá se především softwarovými inženýry a je preferován pro velké, drahé a složité projekty. Projektový manažer určuje počet smyček, které se liší v závislosti na projektu. Každá smyčka spirály je fází v modelu procesu vývoje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



Obrázek 7 – Spirálový model

7. Big Bang

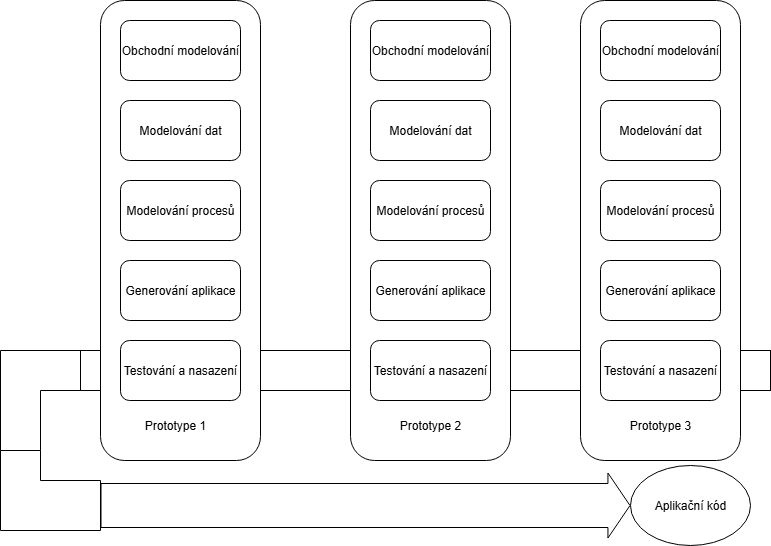
Big Bang je nejjednodušší model, protože nevyžaduje téměř žádné plánování. Vyžaduje však spoustu finančních prostředků a kódování a zabere více času. Tento model kombinuje čas, úsilí a zdroje pro vytvoření produktu. Produkt je postupně vytvářen podle požadavků zákazníka, nicméně konečný produkt nemusí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



Obrázek 8 – Big Bang model

8. RAD (Rapid Application Development)

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.



Obrázek 9 – 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.

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.

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:

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.

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.

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.

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.

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.

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.

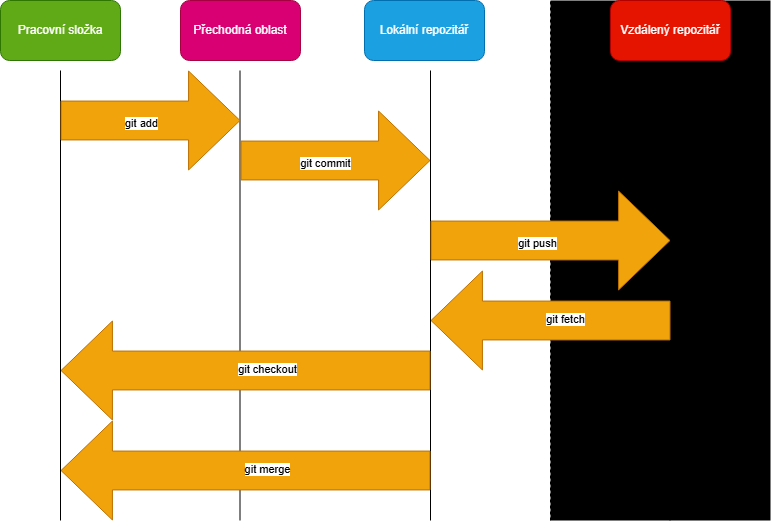
Git také klade velký důraz na integritu dat. Jakákoliv změna či poškození souboru je okamžitě detekováno. Systém ukládá soubory podle kontrolního součtu SHA-1, nikoli podle jména. Tento kontrolní součet provede sám Git pro každý soubor nebo složku.

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.

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:

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 zobrazuje tento proces:



Obrázek 10 – Proces Gitu

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

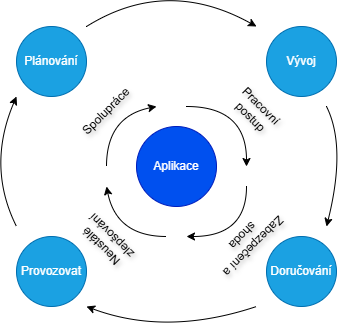
* GitHub
* Bitbucket
* GitLab
* Azure DevOps
* SourceForge
* Beanstalk
* FramaGit

### 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.

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

* 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.
* Vývoj: Fáze vývoje zahrnuje veškeré kódování – od samotného psaní kódu, testování, kontrolu a integraci mezi členy týmu, až po tvorbu build artefaktů.
* 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.
* Provoz: V provozní fázi se vývojáři zaměřují na údržbu a monitorování aplikací a řešení problémů s aplikací, která je již v provozu.



Obrázek 11 – Proces DevOps

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

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

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



Obrázek 12 – 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).

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.

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

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.

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.

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

* Zpřístupnění citlivých dat externím zdrojům
* Použití nezabezpečeného kódu nebo komponent třetích stran
* 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.

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.

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:

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

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.

### 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 více 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.

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

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.

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.

Následuje fáze požadavků, ve které se detailně specifikují bezpečnostní požadavky, provádí se mapování požadavků na bezpečnost a soukromí a je provedena analýza rizik.

Ve fázi návrhu se uplatňují principy bezpečného návrhu a implementují se bezpečnostní kontroly. Klíčovou aktivitou je modelování hrozeb (Threat Modelling). Tato fáze také zahrnuje zapracování zásad a postupů, definování seznamů řízení přístupu, vytváření příručky pro bezpečné kódování a nastavení správy tajemství (Secret Management).

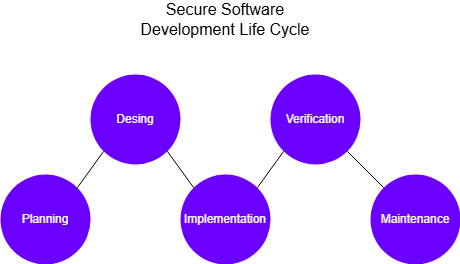
Následuje fáze implementace, která se zaměřuje na dodržování praxe a revizí bezpečného kódu. Zahrnuje integraci DevSecOps nástrojů a metod, jako je SAST (Statická analýza), DAST (Dynamická analýza) a SCA (Anylýza složení software).

Při poslední fázi nasazení je kladen důraz na ověření bezpečnosti před spuštěním, což zahrnuje hodnocení zranitelností, penetrační testování a zajištění bezpečného běhu (Runtime Security).

Celý cyklus uzavírá fáze údržby, která zajišťuje dlouhodobou bezpečnost. Patří sem správa oprav (Patch Management), průběžná údržba zásad a dokumentace a nepřetržité hodnocení a monitorování systému.

Ve fázi návrhu je systematický proces modelování hrozeb. Tento proces využívá hypotetické scénáře, systémové diagramy a testování k zajištění bezpečnosti systémů a dat. Díky identifikaci zranitelností, pomoci s hodnocením rizik a navrhování nápravných opatření přispívá modelování hrozeb ke zlepšení kybernetické bezpečnosti a důvěry.

https://owasp.org/www-chapter-sofia/assets/presentations/202503%20-%20Secure%20Software%20Development:%20Overview%20and%20practical%20examples%20by%20Radostina%20Kondakova.pdf



## 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:

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 efetkivní 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 organitací MITRE Corporation.

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.

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.

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.

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.

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.

Záznamy CVE mohou být ve třech stavech. Stav Reserved (Rezervováno) znamená, že CVE bylo přiřazeno, ale ještě nebylo zvěřejněno. 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, ovčem záznam stále zůstává v záznamu, aby uživatelé věděli, že je neplatný.

Aby bezpečnostní chyba mohla být zapsána, musí splňovat určité podmínky. Jednou z nich je, že chyba musí být opravitelná samostatně bez ohledu na jiné zranitelnost. Dále musí dodavatel uznat, že chyba existuje a negativně ovlivňuje bezpečnost. A také musí chyba ovlivňovat pouze jednu kódovou základnu.

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. 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.

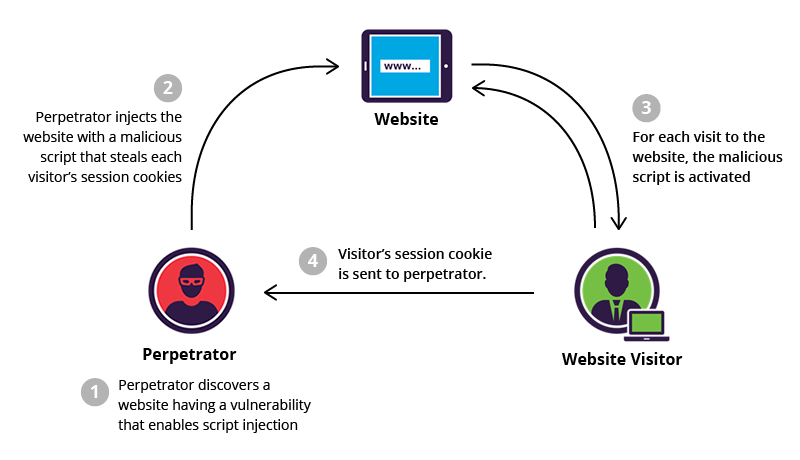
Jednou z nejznámějších zranitelností je CVE-2017-0144. Umožňuje vzdáleným útočníkům spouštět libovolný kód. Známý je především proto, že tato zranitelnost byla zneužita skupinou kyberzločinců v květnu roku 2017. Tento červ se rozšířil na více než 200 000 počítačů ve více než 150 zemích. Mezi známé oběti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

### Kategorizace zranitelností (CWE)

Common Weakness Enumeration (CWE) je seznam běžných slabin software   
a hardwaru. Jedná se o seznam chyb, které mohou vývojáři udělat a které mohou vést k bezpečnostním zranitelnostem. CWE tedy není zranitelnost, je to klasifikace. Ukazuje, jaký druh chybného kódu nebo chování může způsobit bezpečnostní problém.

Seznam CWE spravuje MITRE, která spravuje i seznam CVE. Cílem CWE je, aby vývojáři, bezpečnostní týmy a nástroje mluvili stejným jazykem při identifikaci, diskusi a opravě slabin v kódu.

V seznamu je přes 900 CWE záznamů a jednou z nejvíce nebezpečných slabin je slabina CWE-79. Jedná se o Cross-Site Scripting (XSS) pomocí kterého lze odhalit soukromé informace uložené v cookies uživatele, jako jsou session údaje. To může být obzvlášť nebezpečné, pokud má oběť administrátorská práva pro správu tohoto webu.



Udelat ----- 😊

### 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).

Tento systém byl zaveden v roce 2005 Radou pro národní infrastrukturu USA, Od té doby se systém dál vyvíjel. Před nejnovější verzí byly verze v2, v3.0 a v3.1. Dnes se používá verze v4.0 dostupná od listopadu roku 2023. Tato verze přinesla několik vylepšení jako vetší podrobnosti v základních metrikách, zjednodušení metrik hrozeb s větším dopadem na skóre, doplňkových atributů pro podporu reakce na zranitelnosti. Nejdůležitější ovšem je, že verze 4.0 rozšířila použitelnost na operační technologie (OT), průmyslové řídící systémy (ICS) a prostředí internetu věcí (IoT).

Skóre se získává kombinací několika podskóre. K zařazení zranitelnosti do systému CVSS jsou vyžadovány pouze základní metriky. Pro přesnější hodnocení by skóre mělo obsahovat i environmentální a časové metriky. Celkové CVSS skóre zohledňuje:

* Podskóre dopadu: hodnotí význam poškozených dat a systémů
* Podskóre využitelnosti: hodnotí, jak snadno lze zranitelnost zneužít
* Podskóre rozsahu: hodnotí, jaký je dopad útoku na systémy, které se mohou zdát neovlivněné, a celkový rozsah útoku

Následující tabulka znázorňuje rozdělení jednotlivých úrovní závažnosti podle CVSS verze 4.0:

|  |  |
| --- | --- |
| 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 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.

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

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.

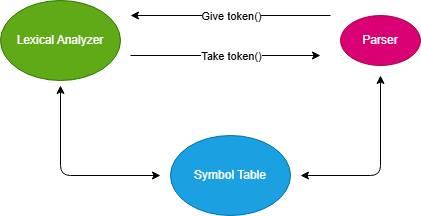
### Statická analýza

Statická analýza hledá zranitelnosti, které jsou náchylné ke kyber útokům. Tato analýza skenuje kód ještě před kompilací. Je také známé jako white box testing, tudíž testování se znalostí kódu, struktury a fungování software. Metoda poskytuje pohled na vnitřní procesy a umožňuje najít konkrétní oblasti, které by mohly obsahovat chyby.

Oproti white box testing existuje i black box testing, kde znalosti vnitřních mechanismů neznáme, tudíž se nemůže jednat o statickou analýzu. Nezaměřujeme se tedy na kód, ale pouze pozorujeme její výkon. Zaměřujeme se na vstupy a výstupy   
a zajišťujeme, aby se software choval podle očekávání.

Používání statické analýzy má několik výhod, zejména pokud je potřeba dodržovat určitý průmyslový standard. Jednou z výhod je rychlost. Ruční kontrola kódu by zabrala spoustu času. Automatizované nástroje jsou mnohem rychlejší. Kromě toho nástroje pro statickou analýzu mají schopnost ověřit všechny možné způsoby, jakými se program může při spuštění chovat. Dokážou prozkoumat všechny části kódu ještě před jeho spuštěním. Během vývoje průběžně kontrolují zdrojový kód a na základě předem definovaných pravidel poskytují detailní přehled o potenciálních problémech, které by se mohly v programu objevit. Dále statická analýza je i přesnější než ruční revize kódu. Při statické analýze je prohledáván každý řádek kódu, aby odhalily možné problémy.

Statická analýza se dělí na lexikální analýzu, syntaktickou analýzu a sémantickou analýzu. Lexikální analýza rozděluje zdrojový kód do sekvence tokenů, jako jsou identifikátory, klíčová slova, operátory a literály. Tento proces je nezbytný pro nástroje, které detekují povrchní problémy se syntaxí, jako jsou neplatné znaky, nesprávně uzavřené řetězce nebo nesprávné použití operátorů. Nástroje pro statickou analýzu většinou používají lexikální analýzu k implementaci základních kontrol, jako jsou konvence pojmenování, pravidla formátování nebo porušení stylu kódu. Tyto kontroly jsou rychlé a mohou zachytit chyby ještě před provedením hlubší analýzy.



### SAST

Tato analýza probíhá v prvních fázích životního cyklu vývoje software. Statickou analýzu zdrojového kódu je možné provádět takto brzy, protože nevyžaduje funkční aplikaci a kód nemusí být vůbec spuštěn.

Nástroje pro statickou analýzu zdrojového kódu poskytují vývojářům okamžitou zpětnou vazbu během psaní kódu. Bezpečnostní problémy jsou tak řešeny zavčas, a ne až na poslední chvíli.

SAST nástroje nabízí grafické zobrazení problémů, od zdrojového kódu po místo, kde by mohly způsobit problém. Některé nástroje dokonce ukazují přesnou pozici zranitelnosti a zvýrazňují rizikový kód. Někdy i poskytují podrobné doporučení, jak problém opravit a s tím i nejlepší místo v kódu, kde opravu provést, aniž by bylo potřeba hluboké bezpečnostní odbornosti.

Dokonce je možné, aby si vývojáři mohli vytvářet vlastní přehledy prostřednictvím SAST nástrojů. Tyto reporty lze exportovat offline a sledovat pomocí dashboardů. Bezpečnostní problémy jsou tak zapsány systematicky a jejich sledování pomáhá vývojářům rychle je řešit a vydávat aplikace s minimálními problémy. Tento proces přispívá k vytvoření bezpečného SDLC.

K efektivnímu provádění statického testování bezpečnosti aplikací je potřeba dodržet šest jednoduchých kroků:

1. Výběr nástroje – Vybrat nástroj pro statickou analýzu, který dokáže provádět kontrolu kódu aplikací napsaných v jazycích, které používáš. Nástroj by měl také rozumět konkrétním frameworkům, na kterých je daný software postaven.
2. Vytvoření infrastruktury a nasazení nástroje – V této fázi je zahrnuto řešení licenčních požadavků, nastavení řízení přístupu a oprávnění, a zajištění potřebných prostředků (např. servery a databáze) pro nasazení nástroje.
3. Přizpůsobení nástroje – Upravení nástroje podle potřeb, tedy například nastavení pravidel pro snížení falešně pozitivních výsledků nebo přidání nové detekce zranitelností.
4. Prioritizace a nasazení aplikací – Pokud je nástroj připraven, může být přidán do aplikace. Pokud je aplikací více je lepší začít u aplikací s nejvyšším rizikem. Postupně by měly být všechny aplikace zahrnuty do pravidelných skenů – ideálně při každé nově vydané verzi, buildu nebo odevzdání kódu.
5. Analýza výsledků skenu – Tato fáze obsahuje třídění výsledků a odstranění falešně pozitivních nálezů. Poté, co je seznam problémů potvrzen, měl by být předán vývojovým týmům k opravě v přiměřeném čase.
6. Správa a školení – Správné řízení zajišťuje, že vývojové týmy používají SAST nástroje efektivně. Bezpečnostní kontroly by měly být pevně integrovány do SDLC.

### Srovnání vybraných nástrojů

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SAST nástroj | Jazyky | Open source | GUI | CI/CD | Generování reportů | Cloud | IDE |
| CodeQL | 12 | Ano | Ano | Ano | Ne | Ano | Ano |
| GitLab SAST | 19 | Ano | Ano | Ano | Ne | Ano | Ano |
| SonarQube | 27 | Ano | Ano | Ano | Ne | Ano | Ano |
| Snyk | 17 | Ano | Ano | Ano | Ne | Ano | Ano |
| Bearer | 10 | Ano? | Ne | Ano | Ne | Ano | Ne |
| Aikido | 16 | Ano | Ano | Ano | Ano | Ano | Ano |
| Horusec | 19 | Ano | Ano | Ano | Ano | Ne | Ano |

## Závěr

Závěr obsahuje stručné shrnutí získaných poznatků, uvedení dalších možných postupů či řešení, hodnocení dostupné odborné literatury, ze které bylo čerpáno. Závěr by měl obsahovat kritické porovnání záměru práce a dosažených výsledků, srovnání dosažených výsledků s dosud známými poznatky a popis odlišností od doposud známých skutečností. Závěr může naznačit praktické uplatnění výsledků práce.

## Závěr

Vytvořená šablona maturitních prací obsahuje formální požadavky maturitních prací na SPŠT Třebíč. Jedná

se zejména o upravené styly v dokumentu, podrobný popis jednotlivých částí maturitní práce a jejího obsahu, snadno editovatelné záhlaví a zápatí s automatickým číslováním stránek a propojení stylů se seznamy a obsahem.

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. CHACON, Scott a STRAUB, Ben. *Getting Started - What is Git?* Online. Pro Git (Second Edition). 2014. Dostupné z: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>. [cit. 2025-08-27].
13. GIT. *Git*. Online. Git-SCM. Dostupné z: <https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F>. [cit. 2025-08-27].
14. 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].
15. BENSON, David. *The Top 10 Version Control Tools*. Online. Logit.io blog. 2024, 04.02.2025. Dostupné z: <https://logit.io/blog/post/version-control-tools>. [cit. 2025-08-27].
16. MICROSOFT. *Co je DevOps?* Online. Dostupné z: <https://azure.microsoft.com/cs-cz/resources/cloud-computing-dictionary/what-is-devops>. [cit. 2025-10-18].
17. 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].
18. THIRMAL, Adhiran. *What is Secure Development?* Online. 2023. Dostupné z: <https://www.securitycompass.com/blog/what-is-secure-development>. [cit. 2025-10-18].
19. FLEXERA. *Software vulnerability*. Online. Dostupné z: <https://www.flexera.com/resources/glossary/software-vulnerability>. [cit. 2025-10-18].
20. RED HAT. *What is a CVE?* Online. 2024. Dostupné z: <https://www.redhat.com/en/topics/security/what-is-cve>. [cit. 2025-10-24].
21. 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].
22. 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].
23. 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].
24. WHATSAPP. *WhatsApp Security Advisories*. Online. 2025. Dostupné z: <https://www.whatsapp.com/security/advisories/2025>. [cit. 2025-10-26].
25. CLOUDFLARE. *What was the WannaCry ransomware attack?* Online. Dostupné z: <https://www.cloudflare.com/learning/security/ransomware/wannacry-ransomware>. [cit. 2025-10-26].
26. MITRE. *What is CWE?* Online. 05. 06. 2023. Dostupné z: <https://cwe.mitre.org/about/new_to_cwe.html>. [cit. 2025-08-27].
27. GOODMAN, Courtney. *What is the Common Vulnerability Scoring System (CVSS)?* Online. Balbix. 25. 10. 2024. Dostupné z: <https://www.balbix.com/insights/understanding-cvss-scores>. [cit. 2025-08-27].
28. FORTINET. *CVSS: Understanding the Common Vulnerability Scores*. Online. Dostupné z: <https://www.fortinet.com/resources/cyberglossary/common-vulnerability-scoring-system>. [cit. 2025-10-26].
29. BLACK DUCK. *Secure Code Review*. Online. Dostupné z: <https://www.blackduck.com/glossary/what-is-code-review.html>. [cit. 2025-08-27].
30. FOSTER, Stuart. Online. 2023. Dostupné z: <https://www.perforce.com/blog/sca/what-static-analysis>. [cit. 2025-10-28].
31. IN-COM. *Co je statická analýza?* Online. 2024. Dostupné z: <https://www.in-com.com/cs/blog/what-is-static-code-analysis/#what-is-static-analysis>. [cit. 2025-08-27].
32. 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. 2025-08-27].
33. GITLAB DOCS. *Static Application Security Testing (SAST)*. Online. Jit. 2025. Dostupné z: <https://docs.gitlab.com/user/application_security/sast>. [cit. 2025-08-27].

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

Seznam obrázků

[Obrázek 1 – SDLC věc 8](#_Toc209087428)

Seznam tabulek

**Nenalezena položka seznamu obrázků.**