

Správa závislostí a prevence útoků na dodavatelský řetězec v softwarovém vývoji

Bc. Petr Zámečník

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	Bc. Petr Zámečník
Osobní číslo:	A23419
Studijní program:	N0613A140022 Informační technologie
Specializace:	Kybernetická bezpečnost
Forma studia:	Kombinovaná
Téma práce:	Správa závislostí a prevence útoků na dodavatelský řetězec v softwarovém vývoji
Téma práce anglicky:	Dependency Management and Prevention of Supply Chain Attacks in Software Development

Zásady pro vypracování

- Provedte analýzu problematiky útoků na dodavatelský řetězec v kontextu softwarového vývoje.
- Identifikujte a popište hlavní typy útoků na dodavatelský řetězec.
- Analyzujte a charakterizujte zranitelnosti spojené se závislostmi v softwarových projektech.
- Zhodnoťte potenciální dopady útoků na dodavatelský řetězec na organizace, včetně finančních ztrát, poškození reputace a narušení provozu.
- Navrhněte strategii pro zmírnění rizik spojených s útoky na dodavatelský řetězec.
- Porovnejte a zhodnoťte přístupy ke správě závislostí v kontextu placeného a open-source softwaru, a navrhněte best practices pro efektivní správu závislostí v obou případech.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. Hammi, Badis & Zeadally, Sherali. (2023). Software Supply-Chain Security: Issues and Countermeasures. Computer. 56. 10.1109/MC.2023.3273491.
2. Mogbeyiteren, Abiloye. (2024). Building secure software: how to avoid security problems the right way.
3. Settas, Dimitris & Stamelos, Ioannis. (2009). Resolving Complexity and Interdependence in Software Project Management Antipatterns Using the Dependency Structure Matrix. 10.1007/978-3-540-70561-1_15.
4. Chen, Jie & Sohal, Amrik & Prajogo, Daniel. (2013). Supply Chain Operational Risk Mitigation: A Collaborative Approach. International Journal of Production Research. 51. 2186-2199. 10.1080/00207543.2012.727490.

Vedoucí diplomové práce: **Ing. David Malaník, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce se zaměřuje na problematiku útoků na dodavatelský řetězec v kontextu softwarového vývoje a význam správy závislostí. Práce analyzuje různé typy útoků na dodavatelský řetězec, včetně dependency confusion, škodlivých závislostí a kompromitace dodavatelů. Zkoumá zranitelnosti spojené se závislostmi, jako jsou zastaralé a špatně spravované závislosti, rizika spojená s open-source a softwarem třetích stran. Práce také hodnotí dopady těchto útoků na organizace, jako jsou finanční ztráty, poškození reputace a narušení provozu. Dále navrhuje strategie zmírnění rizik a osvědčené postupy, včetně bezpečného životního cyklu vývoje softwaru, řízení rizik dodavatelů a transparentnosti dodavatelského řetězce. V neposlední řadě se práce zabývá správou závislostí v kontextu placeného a open-source softwaru.

Klíčová slova: Dodavatelský řetězec, útoky na dodavatelský řetězec, správa závislostí, softwarový vývoj, kybernetická bezpečnost, dependency confusion, zranitelnosti softwaru, open-source software, řízení rizik, životní cyklus vývoje softwaru

ABSTRACT

This thesis focuses on the issue of supply chain attacks in the context of software development and the importance of dependency management. The work analyses various types of supply chain attacks, including dependency confusion, malicious dependencies, and vendor compromises. It examines vulnerabilities associated with dependencies, such as outdated and poorly managed dependencies, risks related to open-source and third-party software. The thesis also evaluates the impacts of these attacks on organizations, including financial losses, reputational damage, and operational disruptions. Furthermore, it proposes risk mitigation strategies and best practices, including secure software development lifecycle, vendor risk management, and supply chain transparency. Lastly, the work addresses dependency management in the context of paid and open-source software.

Keywords: Supply chain, supply chain attacks, dependency management, software development, cybersecurity, dependency confusion, software vulnerabilities, open-source software, risk management, software development lifecycle

Rád bych poděkoval své rodině, za její neochvějnou podporu nejen během vypracovávání této práce, ale také během celého studia, které mě dovedlo až sem.

Mé díky patří také mému vedoucímu práce, Ing. Davidu Malaníkovi, Ph.D., za odborné vedení práce a významné rady, které přispěly k dokončení této diplomové práce.

OBSAH

OBSAH	11
ÚVOD	14
I. TEORETICKÁ ČÁST	15
1 ANALÝZA PROBLEMATIKY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC V KONTEXTU SOFTWARE VÝVOJE	16
1.1 DEFINICE A CHARAKTERISTIKA DODAVATELSKÉHO ŘETĚZCE V SOFTWARE VÝVOJI	16
1.1.1 <i>Fáze softwarového vývoje a integrace externích zdrojů</i>	16
1.1.2 <i>Kritické body vstupu třetích stran</i>	16
1.1.3 <i>Role open-source ekosystému</i>	17
1.1.4 <i>Dodavatelský řetězec jako systémový prvek</i>	17
1.1.5 <i>Geografická a organizační distribuce</i>	17
1.1.6 <i>Legislativní aspekty a standardizace</i>	17
1.1.7 <i>Budoucí trendy a inovace</i>	17
1.2 RIZIKA A HROZBY SPOJENÉ S DODAVATELSKÝM ŘETĚZCEM	18
1.2.1 <i>Kompromitace vývojových nástrojů</i>	18
1.2.2 <i>Zneužití open-source ekosystému</i>	18
1.2.3 <i>Zranitelnosti v závislostech</i>	18
1.2.4 <i>Právní a licenční rizika</i>	18
1.2.5 <i>Geopolitické faktory</i>	18
1.2.6 <i>Nedostatečná transparentnost</i>	19
1.2.7 <i>Ekonomické dopady</i>	19
1.2.8 <i>Faktory zvyšující zranitelnost</i>	19
1.3 PŘEHLED HISTORICKÝCH A SOUČASNÝCH ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC	19
1.3.1 <i>SolarWinds Orion (2020)</i>	20
1.3.2 <i>Codecov Bash Uploader (2021)</i>	20
1.3.3 <i>Log4Shell (2021)</i>	20
1.3.4 <i>Typosquatting v npm (2022-2023)</i>	20
1.3.5 <i>3CX Supply Chain Attack (2023)</i>	20
1.3.6 <i>MOVEit Transfer (2023)</i>	20
1.3.7 <i>XZ Utils Backdoor (2024)</i>	21
2 HLAVNÍ TYPY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC	22
2.1 DEPENDENCY CONFUSION	22
2.1.1 <i>Postup útoku</i>	22
2.1.2 <i>Obranné strategie</i>	24
2.1.3 <i>Případová studie: Útok na Apple</i>	25
2.2 ŠKODLIVÉ ZÁVISLOSTI	26
2.2.1 <i>Zneužití škodlivých závislostí</i>	26
2.2.2 <i>Rizika</i>	26
2.2.3 <i>Prevence a detekce škodlivých závislostí</i>	27
2.3 KOMPROMITACE DODAVATELŮ	28
2.3.1 <i>Mechanismy útoku</i>	28
3 ANALÝZA A CHARAKTERISTIKA ZRANITELNOSTÍ SPOJENÝCH SE ZÁVISLOSTMI V SOFTWARE PROJEKTECH	29
3.1 ZASTARALÉ ZÁVISLOSTI	29
3.2 ŠPATNĚ SPRÁVOVANÉ ZÁVISLOSTI	29
3.3 RIZIKA SPOJENÉ S OPEN-SOURCE SOFTWAREM	30
3.4 RIZIKA SPOJENÉ SE SOFTWAREM TŘETÍCH STRAN	30
4 POTENCIÁLNÍ DOPADY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC	31
4.1 FINANČNÍ ZTRÁTY	32

4.1.1	Přímé náklady incidentu.....	32
4.1.2	Nepřímé dlouhodobé finanční dopady.....	33
4.2	POŠKOZENÍ REPUTACE A ZTRÁTA DŮVĚRY	33
4.2.1	Poškození důvěry klíčových investorů	33
4.2.2	Negativní publicita a mediální pokrytí	34
4.2.3	Potíže při získávání nových zákazníků	34
4.3	NARUŠENÍ PROVOZU	34
4.3.1	Výpadky systémů a nedostupnost služeb.....	34
4.3.2	Zastavení nebo zpomalení vývojových procesů	35
4.3.3	Ztráta nebo poškození dat	35
4.3.4	Přetížení a odklon zdrojů.....	35
4.4	PRÁVNÍ A REGULAČNÍ DŮSLEDKY	36
4.4.1	Pokuty regulačních úřadů	36
4.4.2	Žaloby a soudní spory	36
4.4.3	Povinné audity a zvýšený dohled	37
4.4.4	Porušení smluvních závazků.....	37
4.5	STRATEGICKÉ A KONKURENČNÍ DOPADY	37
4.5.1	Ztráta konkurenční výhody	37
4.5.2	Vliv na fúze a akvizice (M&A).....	38
4.5.3	Dlouhodobý dopad na inovace.....	38
4.6	FAKTORY OVLIVŇUJÍCÍ ZÁVAŽNOST DOPADŮ	38
4.7	PŘÍPADOVÁ STUDIE	39
4.8	ZÁVĚREČNÉ ZHODNOCENÍ	40
II.	PRAKTICKÁ ČÁST	42
5	STRATEGIE PRO ZMÍRNĚNÍ RIZIK ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC	43
5.1	PRINCIPY A CÍLE STRATEGIE ZMÍRNĚNÍ RIZIK	43
5.1.1	Zero Trust.....	43
5.1.2	Defense in Depth.....	44
5.1.3	Shift-Left Security.....	46
5.1.4	Transparentnost a viditelnost	48
5.1.5	Automatizace	50
5.1.6	Hlavní cíle.....	55
5.2	ZABEZPEČENÍ VÝVOJOVÉHO ŽIVOTNÍHO CYKLU	55
5.2.1	Fáze plánování	56
5.2.2	Fáze návrhu.....	56
5.2.3	Fáze implementace	60
5.2.4	Fáze testování.....	60
5.2.5	Fáze nasazení.....	68
5.2.6	Příklad bezpečného Dockerfile pro nasazení.....	68
5.2.7	Fáze údržby	72
5.3	PROAKTIVNÍ SPRÁVA A ZABEZPEČENÍ ZÁVISLOSTÍ.....	73
5.3.1	Význam inventarizace a sledování závislostí.....	73
5.3.2	Generování a správa SBOM	74
5.3.3	Skenování závislostí na známe zranitelnosti	76
5.3.4	Implementace SCA nástrojů	76
5.3.5	Proces prověřování a schvalování nových závislostí	77
5.3.6	Strategie pro pravidelné aktualizace a patchování závislostí	78
5.3.7	Obrana proti útokům typu Dependency Confusion a Typosquatting.....	78
5.3.8	Konfigurace správců balíčků a privátních repositářů	78
5.4	ZABEZPEČENÍ INFRASTRUKTURY A NÁSTROJŮ DODAVATELSKÉHO ŘETĚZCE	80
5.4.1	Zabezpečení systémů pro sestavení a CI/CD pipeline.....	80
5.4.2	Zabezpečení repositářů	82
5.4.3	Zabezpečení vývojářských stanic.....	83
5.5	VERIFIKACE INTEGRITY A PŮVODU SOFTWARE	83
5.5.1	Digitální podepisování	83

5.6	ORGANIZAČNÍ A PROCESNÍ OPATŘENÍ	86
5.6.1	Definování a vynucování bezpečnostních politik.....	86
5.6.2	Pravidelné školení	87
5.6.3	Procesy pro řízení rizik třetích stran (Vendor Risk Management)	87
5.6.4	Vytvoření a testování plánu reakce na incidenty (Incident Response Plan)	87
5.7	PRAKTICKÁ UKÁZKA SCÉNÁŘE REAKCE NA INCIDENT	88
5.7.1	Fáze 0 – Před incidentem	89
5.7.2	Fáze 1 – Detekce a počáteční analýza.....	89
5.7.3	Fáze 2 - Omezení šíření	90
5.7.4	Fáze 3 – Odstranění hrozby.....	91
5.7.5	Fáze 4 – Obnova.....	91
5.7.6	Fáze 5 – Poučení z incidentu	92
5.8	SHRNUTÍ STRATEGIE	92
6	SROVNÁNÍ PŘÍSTUPŮ KE SPRÁVĚ ZÁVISLOSTÍ A NÁVRH NEJLEPŠÍCH PRAKTIK	93
6.1	LICENCOVÁNÍ	93
6.1.1	Komerční software a správa licencí.....	93
6.1.2	Open-source software	95
6.2	PODPORA A ÚDRŽBA.....	96
6.2.1	Podpora a údržba komerčního softwaru.....	96
6.2.2	Podpora a údržba open-source softwaru	96
6.3	TRANSPARENTNOST KÓDU	98
6.3.1	Komerční software	98
6.3.2	Open-source software	98
6.4	CYKLUS VYDÁVÁNÍ NOVÝCH VERZÍ A OPRAVY.....	99
6.4.1	Komerční software	99
6.4.2	Open-source software	99
6.5	NÁKLADY PRO SPRÁVU ZÁVISLOSTÍ.....	100
6.5.1	Komerční software	100
6.5.2	Open-source software	100
6.6	NÁVRH „BEST PRACTICES“ PRO SPRÁVU ZÁVISLOSTÍ.....	101
6.6.1	Kontrola závislostí před nasazením.....	101
6.6.2	Znalost projektových závislostí a SBOM.....	103
6.6.3	Aktivní skenování zranitelností	106
6.6.4	Zabezpečení konfigurace správců balíčků.....	109
6.6.5	Automatizace bezpečnostních kontrol v CI/CD	110
6.6.6	Shift-Left Security	118
6.6.7	Pravidelné aktualizace a patch management.....	124
6.6.8	Závěrečná doporučení a volba nástrojů	125
	ZÁVĚR.....	127
	SEZNAM POUŽITÉ LITERATURY	128
	SEZNAM OBRÁZKŮ	134
	SEZNAM TABULEK	137
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	138
	SEZNAM PŘÍLOH.....	140

ÚVOD

Softwarový vývoj prošel v posledních desetiletích poměrně významnými změnami. Od dob velkých, monolitických aplikací se moderní vývoj posunul spíše k modulární architektuře, kde se software skládá z mnoha menších, navzájem propojených komponent. Tento posun je často spojován s rostoucí komplexitou softwarových systémů a snahou zjednodušení a optimalizování vývojového cyklu.

Mezi klíčové aspekty moderního softwarového vývoje patří široké využití externích komponent a knihoven. Vývojáři stále častěji spoléhají na již existující řešení, které jsou mnohdy na velmi vysoké úrovni. Může se jednat například o kryptografické funkce, webové komponenty nebo databázové operace. Tyto externí komponenty jsou velmi často dostupné jako open-source kód, ale setkáme se také s komerčními produkty.

Tato závislost se však nemusí omezovat pouze na knihovny a komponenty. Moderní softwarový vývoj se také velice často spoléhá na externí dodavatele v různých fázích životního cyklu. To zahrnuje například cloudové platformy pro hostování a provoz aplikací, nástroje pro vývoj a testování, a dokonce i externí týmy vývojářů.

Využití externích zdrojů, ať už se jedná o knihovny, komponenty či vývojáře přináší nepopíratelné výhody v podobě rychlejšího vývoje, vyšší efektivity a někdy i nižších nákladů na vývoj. Nicméně, tato závislost také vytváří nová rizika, zejména v oblasti bezpečnosti. Útoky na dodavatelský řetězec softwaru se stávají stále častější a sofistikovanější, a mohou mít dalekosáhlé dopady na organizace i koncové uživatele.

V kontextu moderního softwarového inženýrství se správa závislostí stává klíčovým prvkem efektivního a bezpečného vývoje. Důkladná správa závislostí umožňuje vývojářům udržet si přehled o všech interních, avšak hlavně externích komponentech, které jsou v projektu použity, a efektivně je aktualizovat a spravovat. Tento proces zahrnuje nejen identifikaci a sledování závislostí, ale také zajištění kompatibility, licencování a v neposlední řadě i bezpečnost. Bez efektivní správy závislostí se projekty stávají náchylnějšími k chybám, konfliktům a bezpečnostním rizikům.

I. TEORETICKÁ ČÁST

1 ANALÝZA PROBLEMATIKY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC V KONTEXTU SOFTWAREVÉHO VÝVOJE

Dodavatelský řetězec v softwarovém vývoji hraje zásadní roli v moderním technologickém ekosystému, kde většina aplikací závisí na externích knihovnách, nástrojích a službách. S rostoucí složitostí těchto řetězců však narůstají i bezpečnostní rizika, která ohrožují integritu softwaru a důvěru uživatelů. Tato analýza se zaměřuje na klíčové aspekty dodavatelského řetězce, včetně jeho definice, charakteristiky a fází zapojení externích dodavatelů, identifikace rizik a hrozeb spojených s těmito závislostmi, a přehledu historických i současných útoků. Cílem je poskytnout ucelený pohled na problematiku a zdůraznit důležitost bezpečnostních opatření v tomto dynamickém prostředí.

1.1 Definice a charakteristika dodavatelského řetězce v softwarovém vývoji

Dodavatelský řetězec v softwarovém vývoji představuje komplexní síť externích komponent, procesů a subjektů, které se podílejí na tvorbě a distribuci softwarových řešení. Tento řetězec zahrnuje nejen kód vytvořený interním týmem, ale i open-source knihovny, cloudové služby, nástroje pro automatizaci a třetí strany poskytující kritické funkce. Vzhledem k rostoucí závislosti na externích prvcích se stal klíčovým prvkem moderního vývoje, kde až 90 % kódu typické aplikace tvoří právě tyto závislosti. [1, 2]

1.1.1 Fáze softwarového vývoje a integrace externích zdrojů

Životní cyklus softwaru se tradičně dělí na fáze plánování, návrhu, implementace, testování, nasazení a údržby. Externí dodavatelé se zapojují již v počátečních fázích – například při výběru vývojových frameworků (React, .NET) nebo nástrojů pro správu závislostí (npm, PyPI, Composer). Během implementace vývojáři integrují open-source balíčky pro řešení specifických problémů, což šetří čas, ale zavádí rizika neprověřeného kódu. [1, 3]

1.1.2 Kritické body vstupu třetích stran

Nejzranitelnějšími místy jsou fáze správy závislostí a aktualizací. Automatizované nástroje jako GitLab CI/CD nebo Jenkins často stahují balíčky přímo z veřejných repositářů bez dostatečné verifikace [3]. Například útok na Codecov v roce 2021 ukázal, jak kompromitace nástroje pro sběr metrik může ovlivnit celou pipeline. Rovněž služby SaaS (např. cloudová

úložiště) představují riziko, jelikož jejich výpadky nebo bezpečnostní chyby paralyzují funkčnost aplikace. [4, 5]

1.1.3 Role open-source ekosystému

Open-source komponenty tvoří páteř moderního vývoje – 85 % podnikových aplikací využívá open-source knihovny. Tyto závislosti však vytvářejí "stínový řetězec", kde vývojové týmy často nemají přehled o všech transitivních závislostech. Studie ukazují, že průměrná aplikace obsahuje přes 500 nepřímých závislostí, z nichž mnohé mohou obsahovat zastaralé nebo zranitelné verze. [2]

1.1.4 Dodavatelský řetězec jako systémový prvek

V kontextu DevOps a CI/CD pipeline se dodavatelský řetězec stal dynamickým organismem, kde změny v jedné komponentě mohou způsobit kaskádové efekty. Microsoft v tomto ohledu zdůrazňuje potřebu tzv. "Chain of Custody" – sledování původu každé komponenty od zdroje po produkční nasazení. Tento přístup vyžaduje implementaci nástrojů pro správu artefaktů (např. JFrog Artifactory) a digitální podpisy kódu. [1, 6]

1.1.5 Geografická a organizační distribuce

Globální povaha dodavatelského řetězce přináší výzvy v podobě rozdílných bezpečnostních standardů mezi regiony. Zatímco evropští dodavatelé často splňují GDPR, asijsí partneři mohou používat nestandardní vývojové praktiky. Případ SolarWinds ukázal, jak útok na jednoho dodavatele v USA může ohrozit vládní instituce po celém světě. [7, 8]

1.1.6 Legislativní aspekty a standardizace

Nové regulace jako americký Executive Order 14028 nebo evropský NIS2 directive explicitně vyžadují správu softwarového dodavatelského řetězce. Tyto předpisy zavádějí povinnost vytvoření Software Bill of Materials (SBOM), který dokumentuje všechny komponenty včetně licencí a známých zranitelností. Implementace těchto norem však naráží na technologickou složitost a náklady. [1]

1.1.7 Budoucí trendy a inovace

Emergentní technologie jako AI-assisted dependency management (např. Dependabot) začínají automatizovat detekci zastaralých balíčků. Současně se prosazuje koncept "Zero Trust Supply Chain", který aplikuje principy mikro segmentace a průběžné verifikace na

všechny komponenty. Tyto přístupy však vyžadují zásadní změnu ve firemní kultuře a procesech. [4, 6]

1.2 Rizika a hrozby spojené s dodavatelským řetězcem

1.2.1 Kompromitace vývojových nástrojů

Útoky na systémy pro sestavení a CI/CD pipeline představují existenční riziko. Případ SolarWinds (2020) demonstroval, jak infikovaná aktualizace Orionu umožnila šíření malwaru do 18 000 organizací včetně vládních agentur. Podobně útok na 3CX (2023) využíval kompromitovaný instalátor komunikačního softwaru k distribuci bankovního malwaru. [6, 8]

1.2.2 Zneužití open-source ekosystému

Kyberzločinci stále častěji zneužívají veřejné repositáře k distribuci škodlivého kódu. V letech 2022-2023 došlo k 633% nárůstu typosquattingových útoků na npm a PyPI, kde balíčky s chybně napsanými názvy imitují populární knihovny. Například balíček "crossenv" místo "cross-env" obsahoval ransomware zaměřený na Node.js aplikace. [2]

1.2.3 Zranitelnosti v závislostech

I legitimní komponenty mohou obsahovat kritické chyby. Případ **Log4Shell** (2021) v logovací knihovně **Log4j** ovlivnil přes 70 % podnikových sítí, protože zranitelnost umožňovala vzdálené spuštění kódu.[2] Podobně chyba v knihovně **OpenSSL (Heartbleed, 2014)** ohrozila šifrování na milionech serverů. [5]

1.2.4 Právní a licenční rizika

Nekontrolované použití GPL komponent může vést k povinnosti zveřejnit vlastní kód. Studie ukazují, že 33 % společností neprovádí pravidelný audit licencí, což vystavuje riziku žalob. Případ společnosti Versata (2015) proti Americké armádě ukázal, jak licenční spory mohou zablokovat dodávky kritického softwaru. [5]

1.2.5 Geopolitické faktory

Koncentrace dodavatelů v určitých regionech vytváří strategická rizika. Například 78 % open-source přispěvatelů pochází z Číny a USA, což v kontextu obchodních válek ohrožuje

kontinuitu dodávek. Zákaz Huawei v 5G sítích některých zemí ilustruje dopady politických rozhodnutí na dodavatelské řetězce. [2, 9]

1.2.6 Nedostatečná transparentnost

Neprůhledné dodavatelské vztahy komplikují reakci na incidenty. Průzkum uvádí, že 65 % společností nemá kompletní mapu svých závislostí. Tento problém vyvrcholil při útoku na Kaseya (2021), kdy kompromitace jednoho MSP vedla k šíření ransomwaru do 1 500 firem. [5, 10]

1.2.7 Ekonomické dopady

Průměrné náklady na řešení incidentu v dodavatelském řetězci přesahují 4,5 milionu USD. Vedle přímých nákladů na odstranění malwaru společnosti čelí pokutám za GDPR porušení a ztrátě důvěry zákazníků. Případ britského Electoral Commission (2023), kde únik dat 40 milionů voličů vedl k 20% poklesu účasti ve volbách, ukazuje širší společenské důsledky. [2, 9]

1.2.8 Faktory zvyšující zranitelnost

Složitost moderních aplikací s mikro službami exponenciálně zvyšuje plochu útoku Studie SAP uvádí, že průměrný podnik spravuje 15 000 závislostí napříč 850 různými repositáři. K tomu přispívá i trend "shift left", kdy vývojáři integrují externí komponenty již v raných fázích vývoje bez dostatečného zabezpečení.[6]

1.3 Přehled historických a současných útoků na dodavatelský řetězec

Historické útoky poukazují na rizika útoků na dodavatelský řetězec a zdůrazňují tak potřebu:

- Implementace Software Bill of Materials (**SBOM**) pro mapování závislostí
- Používání sigstore pro kryptografické podepisování artefaktů
- Pravidelné penetrační testování komponent třetích stran
- Zavedení Zero Trust architektury v CI/CD pipelinech
- Vzdělávání vývojářů v oblasti kybernetické bezpečnosti

1.3.1 SolarWinds Orion (2020)

Kompromitace sestavovacího systému SolarWinds umožnila distribuci backdooru SUNBURST skrze legitimní aktualizace. Útočníci ze skupiny Nobelium infiltrovali síť 100+ společností včetně Microsoftu a vládních agentur. Lekcí je potřeba izolace build prostředí a implementace vícefaktorové autentizace pro kritické systémy. [11]

1.3.2 Codecov Bash Uploader (2021)

Manipulace s nástrojem pro sběr testovacích metrik umožnila krádež přihlašovacích údajů z CI/CD pipeline. Útok postihl 29 000 zákazníků včetně HashiCorp a Procter & Gamble. Důraz na ověřování integrity skriptů a omezení přístupu k citlivým údajům v pipeline se stal klíčovým opatřením. [5]

1.3.3 Log4Shell (2021)

Zneužití zranitelnosti CVE-2021-44228 v knihovně Log4j demonstrovalo rizika všudypřítomných open-source komponent. Útoky zasáhly od cloudu AWS po herní servery Minecraft, s odhadovanými globálními náklady 10 miliard USD. Výsledkem bylo masivní nasazení nástrojů pro automatické skenování závislostí (SCA). [2, 3]

1.3.4 Typosquatting v npm (2022-2023)

Balíčky jako "crossenv" nebo "marak-url-parser" obsahující ransomware ukázaly zneužití důvěry ve veřejné repositáře. Útoky vynutily zavedení dvoufázového ověřování pro správce a automatickou detekci typosquattingu. [2, 5]

1.3.5 3CX Supply Chain Attack (2023)

Kompromitace komunikační platformy 3CX prostřednictvím infikované knihovny vedla k distribuci malware Gopuram. Útok spojený se severokorejskou skupinou Lazarus zasáhl 600 000 organizací. Lekcí je nutnost skenování všech třetích komponent včetně ikon a mediálních souborů.

1.3.6 MOVEit Transfer (2023)

Zneužití zero-day zranitelnosti v MOVEit společnosti Progress Software umožnilo exfiltraci dat z 2 500 organizací včetně British Airways a BBC. Skupina Cl0p vydírala oběti kombinací ransomware a DDoS útoků. Incident urychlil adopci exploit prediction scoring systémů (EPSS). [6]

1.3.7 XZ Utils Backdoor (2024)

Sofistikovaná infiltrace open-source projektu XZ ukázala rizika sociálního inženýrství. Útočník pod falešnou identitou "Jia Tan" implantoval backdoor do kompresní knihovny, což ohrozilo distribuce Linuxu. Tento případ vedl k revizi procesů pro přijímání externích přispěvatelů v kritických projektech. [12]

2 HLAVNÍ TYPY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC

Hlavní typy útoků na dodavatelský řetězec využívají zejména rostoucí složitosti softwarových ekosystémů a globalizace dodavatelských řetězců. Útoky se zaměřují na slabá místa v těchto strukturách a stávají se stále závažnější hrozbou digitální doby.

2.1 Dependency Confusion

Dependency confusion – neboli záměna závislostí je typ útoku na dodavatelský řetězec, kde útočník nahraje škodlivý balíček se stejným názvem jako využívá interní balíček organizace do veřejného repositáře (např. npm, pip, Maven). Pokud je správce balíčků špatně nastaven, může namísto interního balíčku nainstalovat právě ten škodlivý, což umožní útočníkovi spustit škodlivý kód a docílit tak například krádeže dat nebo kompromitace systému.

Útok spoléhá na fakt, že mnoho organizací po celém světě používá jak interní, tak veřejné repositáře pro správu balíčků neboli závislostí. Tento útok je obzvláště nebezpečný, protože umožňuje vzdálené spuštění kódu a potenciální kompromitaci celého vývojového prostředí.

Například v roce 2021, výzkumník Alex Birsan úspěšně napadnul více než 35 velkých společností, mezi které patří například Apple a Microsoft. Díky tomu získal odměnu za bug bounty v hodnotě až 130 000 amerických dolarů. Toto pouze dokládá, jak rozšířené mohou tyto zranitelnosti opravdu být. [13]

2.1.1 Postup útoku

Proces útoku je možné rozdělit do několika fází. Tento typ útoků nemusí nutně vyžadovat vysoce sofistikované technické znalosti ze strany útočníka, ale spíše spoléhá na chybnou konfiguraci, nebo neopatrnost ze strany cílové organizace.

a) Průzkum

Prvním krokem útočníka je identifikace názvů interních balíčků, které cílová organizace používá. Získat je může několika způsoby. Často prohledává veřejné repositáře, jako je GitHub, kde mohou být nechtěně zveřejněny konfigurační soubory (např. package.json, requirements.txt) obsahující tyto názvy [14].

Další možností je analýza otevřených zdrojových projektů, které jsou s organizací spojeny. V úvahu připadá také vyhledávání úniků interní dokumentace

nebo využití sociálního inženýrství, jako je phishing či spearphishing zaměstnanců, k získání potřebných informací.

K nalezení názvů závislostí může útočník například použít tzv. GitHub dorking s podobným dotazem:

```
path:**/package.json org:název-organizace
```

b) Vytvoření škodlivého balíčku

Po identifikaci názvu interního balíčku útočník vytvoří škodlivý balíček se stejným názvem. Tento balíček obsahuje škodlivý kód navržený tak, aby se spustil při instalaci nebo prvním použití. Útočník může například v npm využít skript preinstall ke spuštění kódu ihned po instalaci, nebo v Pythonu zahrnout škodlivé příkazy do souboru setup.py, které mohou třeba odesílat systémové informace na server útočníka.

Klíčovým prvkem této fáze je vydání škodlivého balíčku s vyšším (nebo stejným, ale novějším) číslem verze, než jaké má legitimní interní balíček. Tím útočník zajistí, že nesprávně nakonfigurovaný správce balíčků upřednostní škodlivou verzi. Pokud systémy instalují nejnovější dostupné verze. Bez adekvátní kontroly původu či zabezpečení se tato metoda stává pro útočníka velmi snadnou cestou do systémů cílové organizace.

c) Nahrání do veřejného repositáře

Následně útočník nahraje vytvořený škodlivý balíček do veřejného repositáře, jako je npm nebo PyPI. Musí zajistit, že balíček je veřejně dostupný a instalovatelný kýmkoliv. Pokud je daný název balíčku ve veřejném repositáři již obsazen, útočník je nucen zvolit jinou strategii.

d) Čekání na instalaci

V této fázi útočník vyčkává, až si cílový systém (např. CI/CD pipeline) nebo vývojář škodlivý balíček nainstaluje. K tomu může dojít za různých okolností. Častou příčinou je nesprávná konfigurace správce balíčků, který při hledání závislostí upřednostní veřejný repositář a vyšší verzi balíčku. Instalace může proběhnout také v situaci, kdy je interní repositář dočasně nedostupný, například když vývojář pracuje

z domova bez připojení k firemní síti. Nakonec může k instalaci dojít i omylem, pokud vývojář nespecifikuje přesnou verzi nebo zdroj balíčku.

e) Zneužití škodlivého kódu

Jakmile je škodlivý balíček nainstalován a případně aktivován (např. importován v kódu), útočník může spustit svůj škodlivý kód. Cíle mohou být různé, ať už exfiltrace citlivých dat, například pomocí skrytých DNS dotazů [15], kompromitace systému instalací trvalejších zadních vrátek – tzv. backdoor, pro budoucí přístup, nebo spuštění dalších destruktivních útoků, jako je nasazení ransomware či šíření malwaru v rámci napadené sítě.

2.1.2 Obranné strategie

Aby organizace zabránila útokům typu Dependency Confusion, je nezbytné implementovat strategii, která kombinuje technická opatření s procesními změnami. Důležité je si uvědomit, že obrana před těmito útoky vyžaduje neustálou ostražitost.

a) Správná konfigurace správce balíčků

Správná konfigurace správce balíčků vyžaduje nastavení preference pro interní repositář a zákaz automatického vyhledávání ve veřejných repositářích pro interní balíčky. Například v Pythonu se doporučuje vyhnout použití parametru „—extra-index-url“, který by mohl vést k instalaci z PyPI místo z interního zdroje [16]. Současně musí organizace zajistit přístup vývojářů k internímu repositáři, a to i při práci z domova.

b) Použití názvosloví – Namespacing

Používání specifického názvosloví či oborů názvů (tzv. scopes) pro interní balíčky pomáhá vývojářům vyhnout se konfliktům s veřejnými balíčky. Například v npm formát „@společnost/balíček“ zajistí, že se balíček bude hledat primárně v interním repositáři spojeném s daným oborem názvů [17].

c) Rezervace názvů balíčků

Další obrannou technikou je rezervace názvů interních balíčků ve veřejných repositářích, aby útočníci nemohli nahrát škodlivé balíčky se stejnými názvy. Některé repositáře tuto rezervaci přímo umožňují. Alternativně lze publikovat prázdný balíček se stejným názvem, který tento název obsadí [18].

d) Verzování závislostí

Specifikování přesných čísel verzí v konfiguračních souborech závislostí, například „package.json“ brání automatickým aktualizacím na novější, potenciálně nebezpečné verze. Tato praxe známá také jako „pinnin“ verzí, představuje klíčový prvek pro zajištění konzistence i bezpečnosti balíčků [19]

e) Monitorování a auditování

Je vhodné pravidelně monitorovat veřejné repozitáře na přítomnost balíčků se stejnými názvy, jako interní balíčky a případně okamžitě reagovat na podezřelé aktivity [20].

f) Vzdělávání zaměstnanců

Klíčovým bezpečnostním opatřením zůstává také vzdělávání zaměstnanců, zejména vývojářů, o rizicích spojených s Dependency Confusion a o správném nastavení správce balíčků. Vývojáři musí rozumět rozdílu mezi interními a veřejnými balíčky a vědět, jak bezpečně konfigurovat svá vývojová prostředí [21].

g) Použití soukromých repozitářů

Všechny interní balíčky by měly být uchovávány v soukromých repozitářích, které nejsou přístupné veřejnosti. Zároveň je třeba zajistit, aby vývojáři pro interní závislosti používali výhradně tyto repozitáře.

h) Bezpečnost CI/CD pipeline

Ověřování závislostí přímo v CI/CD pipelinech zajišťuje, že pocházejí z důvěryhodných zdrojů. Proto je nutné implementovat vhodné kontroly, které detekují a blokují instalaci interních balíčků z veřejných repozitářů během automatizovaných procesů [14].

2.1.3 Případová studie: Útok na Apple

Alex Birsan ve svém výzkumu v roce 2021 úspěšně napadl Apple nahráním škodlivého npm balíčku, který byl spuštěn na systémech spojených s Apple ID. Apple problém rychle vyřešil a udělil Birsanovi odměnu ve výši 30 000 amerických dolarů. Tento případ ukazuje, jak mohou být velké organizace zranitelné, pokud nejsou správně nastaveny jejich systémy pro správu balíčků a závislostí. [13]

2.2 Škodlivé závislosti

Škodlivé závislosti jsou softwarové balíčky, které úmyslně obsahují škodlivý kód, nebo jsou navrženy tak, aby poškodily systém při instalaci nebo jejich použití. Může se jednat nově vytvořené balíčky, nebo již existující balíčky, které byly kompromitovány, například napadením úložiště. Tyto závislosti mohou vést ke vzdálenému spuštění kódu, krádeži dat, kompromitaci systému nebo narušení služeb. Příkladem může být například incident s knihovnou EventStream v roce 2018, kdy byl kompromitován balíček a došlo ke krádeži Bitcoinů [22]

2.2.1 Zneužití škodlivých závislostí

Útočníci mohou zavést škodlivé závislosti několika způsoby, například:

1. **Vytvoření nových škodlivých balíčků:** Útočníci mohou vytvořit balíčky, které vypadají legitimně, ale obsahují škodlivý kód. Často se tak děje pomocí typosquattingu, kde jsou názvy balíčků podobné známým, populárním balíčkům, čímž mohou oklamat vývojáře k instalaci špatného balíčku. Nemusí se však jednat pouze oklamáním, ale také nešťastnou náhodou, kdy se vývojář napíše jméno balíčku špatně, a místo legitimního tak nainstaluje škodlivý.
2. **Kompromitace existujících balíčků:** Útočník může získat kontrolu nad již existujícím, populárním balíčkem, ať už napadením účtů spravující daný balíček, nebo případným zneužitím zranitelností v systému pro správu balíčků a poté aktualizovat balíček tak, aby obsahoval škodlivý kód.
3. **Útok na řetězec závislostí:** Pokud má balíček vlastní závislosti, škodlivý balíček hluboko ve stromu závislostí může ovlivnit všechny aplikace, které na něm nepřímo závisí

2.2.2 Rizika

Rizika spojená se škodlivými závislostmi jsou značná, protože mohou vést k:

- Vzdálenému spuštění kódu
- Krádeži dat
- Kompromitaci systému
- Narušení služeb, či chodu systému

2.2.3 Prevence a detekce škodlivých závislostí

Škodlivé závislosti představují stále rostoucí hrozbu pro bezpečnost softwarových systémů, a proto je nezbytné přijmout komplexní opatření k jejich prevenci a včasné detekci. Tato opatření by měla zahrnovat jak technické prostředky, tak procesní změny, aby organizace minimalizovaly riziko zavedení škodlivého kódu prostřednictvím balíčků.

Prvním zásadním krokem je využívání specializovaných nástrojů pro skenování závislostí, které dokážou odhalit známe zranitelnosti a přítomnost škodlivého kódu. Mezi oblíbené nástroje patří například Snyk, Anchore nebo Black Duck, které umožňují automatizované analýzy balíčků a poskytují aktuální informace o jejich bezpečnostním stavu. Tyto nástroje by měly být integrovány do vývojového cyklu – ideálně již od fáze návrhu až po nasazení – aby bylo možné včasné identifikovat potenciální hrozby. Důležité je také zajistit pravidelné monitorování, jelikož nové zranitelnosti mohou být objeveny kdykoliv po vydání balíčku. Tímto způsobem lze předejít situacím, kdy by škodlivý kód zůstal bez povšimnutí. [23]

Dalším klíčovým opatřením je ověřování autenticity a integrity balíčků před jejich instalací, aby bylo zajištěno, že nebyly kompromitovány. Toho lze dosáhnout například kontrolou digitálních podpisů, nebo výpočtem a porovnáním tzv. checksumů, které potvrzují, že stažený balíček odpovídá původnímu zdroji. Současně je nutné pravidelně aktualizovat závislosti na nejnovější bezpečné verze, čímž se snižuje riziko zneužití známých zranitelností. Pro efektivní správu aktualizací se doporučuje vést tzv. Software Bill of Materials (SBOM), což je strukturovaný seznam všech použitých závislostí a jejich verzí. Standardy jako SPDX nebo CycloneDX mohou tento proces usnadnit a umožnit rychlou identifikaci zastaralých, či rizikových závislostí. Kombinace těchto postupů pomáhá eliminovat riziko instalace škodlivých balíčků. [24]

Nakonec je nezbytné implementovat důkladný proces analýzy celého stromu závislostí, který zahrnuje nejen přímo použité balíčky, ale i jejich vlastní nepřímé závislosti. Tento přístup umožňuje odhalit bezpečnostní rizika i v situacích, kdy je zranitelná či škodlivá pouze hlubší úroveň závislostního řetězce, což může být jinak přehlédnuto. K identifikaci těchto rizik lze efektivně využít porovnání informací o balíčcích s databázemi známých zranitelností, jako je například NVD (National Vulnerability Database), případně doplnit analýzu o specializované nástroje, které zkoumají bezpečnostní historii a původ jednotlivých komponent. [25]

2.3 Kompromitace dodavatelů

Kompromitace dodavatelů je typ útoku na dodavatelský řetězec, při kterém útočník získá neoprávněný přístup k systémům, procesům nebo produktům dodavatele, aby tyto kompromitoval a využil je k útoku na zákazníky dodavatele. V softwarovém vývoji se může jednat o manipulaci aktualizací nebo vložení škodlivého kódu v podobě backdooru. Tento typ útoku je považován za obzvlášť nebezpečný, protože zneužívá důvěry mezi dodavatelem a jeho zákazníky, což umožňuje útočníkům obcházet tradiční bezpečnostní opatření.

2.3.1 Mechanismy útoku

Útočníci mohou využít různé metody ke kompromitaci dodavatelů, některé z nich jsou obzvláště zákeřné.

Při narušení vývojového prostředí mohou útočníci získat přístup k interním systémům dodavatele, jako jsou systémy pro sestavování (build) softwaru a vložit škodlivý kód přímo do finálních produktů. Například v případě SolarWinds byl kompromitován proces sestavování, což vedlo k distribuci škodlivých aktualizací.[26]

Jako další mechanismus je možné využít sociálního inženýrství, kde útočník využije například phishing, spear-phishing, nebo jiné techniky sociálního inženýrství k získání přístupu k účtům dodavatele, což jim umožní manipulaci s produkty či daty zákazníka. Cílem tohoto útoku se nedávno stalo herní studio Grinding Gear Games, přičemž útočník kompromitoval starý Steam účet pro testování, čímž získal administrátorský přístup a změnil hesla na 66 hráčských účtech. Hráči přišli o různé předměty ve hře Path of Exile 2, ale také byly vyzrazeny jejich osobní informace, jako e-mailové adresy, Steam ID, IP adresy a případně také jejich doručovací adresy. [27]

V některých případech mohou útočníci manipulovat přímo s hardwarovými komponenty během jejich výroby nebo distribuce, což je méně časté, ale stále možné. Tato mechanika je velmi nebezpečná, jelikož může být téměř nemožné odhalit potenciálně napadené kusy hardwaru. [28]

3 ANALÝZA A CHARAKTERISTIKA ZRANITELNOSTÍ SPOJENÝCH SE ZÁVISLOSTMI V SOFTWARECH PROJEKTECH

Moderní softwarové projekty jsou velmi často úzce spjaty s využíváním velkého množství externích závislostí, ať už se jedná o open-source knihovny, frameworky nebo komerční softwarové komponenty. Tato praxe sice urychluje vývojový cyklus, avšak současně představuje významnou vstupní bránu pro různé typy zranitelností.

3.1 Zastaralé závislosti

Zastaralé závislosti představují knihovny nebo balíčky, které nejsou aktualizovány a často obsahují známe zranitelnosti. Tyto zranitelnosti mohou zahrnovat neopravené CVE, což jsou známe bezpečnostní chyby a zastaralé funkce, které již nejsou podporované kvůli bezpečnostním rizikům. Například RCE – Remote Code Execution umožňuje útočníkům spustit libovolný kód na systému, což může vést k úplnému kompromitování systému.

Hlavním problémem zastaralých závislostí je zejména veřejná informace o daných zranitelnostech, v případě že vývojář či organizace využívají zranitelné závislosti, může být relativně snadné zneužít této informace. To se například stalo v případě útoku na společnost Equifax v roce 2017. Zde byla zranitelnost v Apache Struts (CVE-2017-5638), která útok umožnila, ten vedl k úniku citlivých dat přibližně 143 milionů zákazníků. [29]

Další příklady zahrnují například SQL Injection, kde zastaralé knihovny nemusí správně validovat vstupy, což umožňuje injektovat škodlivé SQL dotazy a XSS, kde útočník může vložit JavaScript kód. To může vést k odcizení přihlašovacích údajů.

Statistiky ukazují, že v roce 2018 bylo objeveno přibližně 16 000 nových CVE, což zdůrazňuje rostoucí problém se zastaralými závislostmi [30]. Tento trend zvyšuje riziko pro různé programovací ekosystémy, jako je NPM, kde zastaralé balíčky mohou být snadno zneužity kvůli vysoké závislosti na komunitních balíčcích.

3.2 Špatně spravované závislosti

Špatně spravované závislosti zahrnují praktiky, jako je nedostatek aktualizací, nesprávné verzování a ignorování bezpečnostních upozornění. Tyto praktiky pak vedou k použití zastaralých knihoven se známými chybami, což zvyšuje riziko zneužití.

Dopad na celkovou bezpečnost má například vyšší riziko u známých zranitelností, obtížnost identifikace například když je zranitelná právě nepřímá komponenta a také obtížnost opravy zranitelností, například v případě použití závislostí, které již nejsou udržované a aktualizované.

Statistiky ukazují, že v testovaných aplikacích až 86 % z nich obsahovalo alespoň jednu známou zranitelnost v open-source komponentech. Přitom v 81 % případů se jednalo o zranitelnost s vysokým, až kritickým rizikem. Nejčastější zranitelnosti s vysokým rizikem byly nalezeny v komponentách jQuery. [31]

3.3 Rizika spojené s open-source softwarem

Open source software přináší výhody, ale také specifická rizika, jako je například nedostatek údržby, což vede k neopraveným zranitelnostem a možnost vložení malwaru útočníky přispívajícími do open source projektů. Další rizika zahrnují například typosquatting, kde útočníci vytvářejí balíčky s podobnými názvy a zneužívají nepozornosti či chyb vývojářů při instalaci balíčků.

3.4 Rizika spojené se softwarem třetích stran

Software třetích stran, zejména proprietární řešení, přináší výzvy kvůli omezené transparentnosti, což ztěžuje auditování potenciálních zranitelností. Další problém je závislost na podpoře dodavatele pro bezpečnostní aktualizace, což může vést k pomalým opravám bezpečnostních rizik.

Strategie pro zmírnění rizik zahrnují pravidelné auditování dodavatelů, kde se zkoumají jejich bezpečnostní praktiky a historie, případně bezpečnostní záruky ve smlouvách, které vyžadují včasné aktualizace, monitorování a auditování softwaru pro včasnou identifikaci a opravu potenciálních bezpečnostních rizik. Jako příklad je možné zmínit útok na SolarWinds, kde kompromitovaná aktualizace Orion softwaru ovlivnila mnoho organizací po celém světě [8, 26].

Mezi moderní trendy patří například zavádění SBOM (Software Bill of Materials) pro lepší sledování závislostí a rizik v použitých komponentách třetích stran.

4 POTENCIÁLNÍ DOPADY ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC

Útoky na softwarový dodavatelský řetězec představují vážnou a často podceňovanou hrozbu. Jedná se o kompromitaci softwaru v kterékoliv fázi jeho životního cyklu. Kompromitovaný software pak může způsobit závažné problémy, jak organizaci, která byla napadena, tak její uživatelům.

Závažnost těchto útoků pramení z několika faktorů. Moderní software se zřídka vyvíjí zcela od základu. Místo toho se spoléhá na množství externích komponent, jako jsou API třetích stran, open-source kód a proprietární knihovny od dodavatelů. Průměrný projekt dnes obsahuje více než 200 závislostí. Pokud populární aplikace obsahuje, byť jen jedinou kompromitovanou závislost, je ohrožena každá organizace, která ji používá, což vede k exponenciálnímu nárůstu potenciálních obětí. [32].

Útoky na dodavatelský řetězec patří mezi nejobtížněji odvratitelné hrozby, protože zneužívají důvěry v rámci celého ekosystému. Často zahrnují vložení zadních vrátek do běžně používaného softwaru, což útočníkům umožňuje distribuovat škodlivé aktualizace, které mohou obejít standardní bezpečnostní opatření. Jelikož tyto aktualizace pocházejí z důvěryhodných zdrojů, tradiční bezpečnostní nástroje je nemusí detekovat jako hrozby. Dopady tak mohou být navíc dlouhodobé, což komplikuje jejich zmírnění a následně nápravu [33].

Podceňování těchto útoků je často způsobeno omezeními tradičních metod skenování zranitelností, které neposkytují komplexní přehled o rizicích spojených se softwarovými komponentami a závislostmi. Mnoho organizací postrádá detailní přehled o svých softwarových aktivech a závislostech, často nemají k dispozici ani tzv. Software Bill of Materials (SBOM) [34]. Pozornost se také často soustředí na zabezpečení frontendových aplikací, zatímco zranitelnosti v dalších částech dodavatelského řetězce, jako jsou CI/CD pipeline, servery a vývojářské nástroje mohou být přehlíženy [35].

Důležité je zdůraznit, že dopady útoků na softwarový dodavatelský řetězec nejsou izolované. Jsou vzájemně propojené a mohou vyvolat kaskádový efekt. Kompromitace jediného dodavatele může spustit řetězovou reakci bezpečnostních incidentů napříč mnoha organizacemi. Každý článek v tomto řetězci představuje potenciální zranitelnost, kterou mohou útočníci zneužít. Jediný kybernetický útok nebo softwarová zranitelnost může zastavit celé dodavatelské řetězce. Následné narušení provozu může vést k finančním

ztrátám a poškození reputace, což může snížit budoucí příjmy a ztížit získávání nových zákazníků. Finanční ztráty mohou vést ke zvýšení nákladů na pojištění a potenciálně ovlivnit ceny akcií. Bezpečnostní incident může také vyvolat regulační kontrolu a právní kroky, což dále prohlubuje finanční a reputační škody. Vzájemně propojená povaha moderních dodavatelských řetězců znamená, že dopad útoku na jednu organizaci či entitu, se může rozšířit na další, které na ni spoléhají, čímž vzniká dominový efekt [36, 37].

4.1 Finanční ztráty

Útoky na softwarový dodavatelský řetězec mohou způsobit značné finanční ztráty, které lze rozdělit na přímé náklady spojené s řešením incidentu a nepřímé, dlouhodobé finanční dopady

4.1.1 Přímé náklady incidentu

Přímé náklady incidentu zahrnují několik klíčových oblastí. První z nich jsou **náklady na reakci na incident**, které pokrývají okamžité kroky k omezení, analýze a řešení incidentu. Klíčovou součástí je forenzní analýza pro pochopení rozsahu a příčiny útoku, přičemž náklady na ni se mohou pohybovat přibližně od 1600 amerických dolarů až po více než 100 000 amerických dolarů [38, 39].

Další významnou položkou jsou **náklady na obnovu a nápravu**. Ty zahrnují čištění infikovaných systémů, odstranění malwaru a obnovu dat ze záloh, což může být komplikované, pokud jsou zálohy taktéž kompromitovány. Průměrné náklady na při narušení dat zahrnující kompromitaci softwarového dodavatelského řetězce činily v roce 2023 přibližně 4,63 milionů amerických dolarů [40]. Nezbytná je také implementace nových bezpečnostních opatření pro prevenci budoucích útoků. Fáze obnovy může být nákladnější než počáteční reakce a vyžaduje značné investice do technologií, procesů a školení.

Specifickým případem jsou náklady spojené s **výkupným v případě ransomwarových útoků**, které jsou na vzestupu, zejména v oblasti dodavatelského řetězce [41]. Průměrná platba výkupného v roce 2024 dosáhla 2 milionů amerických dolarů [42]. Placení výkupného představuje dilema, protože nezaručuje obnovu dat a může povzbudit k dalším útokům. Pouze 47 % platících získalo nepoškozená data zpět a 78 % bylo znovu napadeno [43].

K přímým nákladům patří rovněž **náklady na komunikaci**. Informování postižených stran, jako jsou zákazníci, partneři a regulátoři, je zákonnou a etickou povinností [44].

Nakonec nelze opomenout **právní poplatky a náklady na soudní spory**. Právní poplatky spojené s poradenstvím, regulačními šetřeními a potenciálními soudními spory ze strany zákazníků a partnerů mohou představovat významné náklady. Hromadné žaloby jsou běžné a mohou vést k vyrovnáním v řádech milionů až miliard dolarů. Právní důsledky mohou přetrvávat roky po samotném incidentu [33, 44].

4.1.2 Nepřímé dlouhodobé finanční dopady

Kromě přímých nákladů existují i významné nepřímé dopady. Jedním z nejcitelnějších je **ztráta příjmů**, která je důsledkem narušení provozu, výpadků systémů a nemožnosti provozovat běžnou činnost. Průměrné náklady na výpadky mohou dosahovat velmi vysokých částek, přičemž v kritických odvětvích mohou šplhat až k milionům amerických dolarů za hodinu [37].

Dalším důsledkem je **odchod zákazníků**. Ztráta důvěry zákazníků po bezpečnostním incidentu často vede k jejich přechodu ke konkurenci, kterou mohou vnímat jako bezpečnější. Současně se stává náročnějším získávat nové zákazníky. Dlouhodobý dopad ztráty důvěry zákazníků negativně ovlivňuje jak současné, tak budoucí příjmy.

Bezpečnostní incidenty mají také dopad na finanční náklady spojené s pojištěním. Dochází ke **zvýšení nákladů na pojištění kybernetických rizik**, protože pojistné roste v reakci na zvyšující se frekvenci a závažnosti útoků. Organizace které prodělaly incident, mohou čelit výrazně vyšším pojistným sazbám [45].

Pro veřejně obchodované společnosti může incident znamenat **pokles hodnoty akcií**. Narušení dat má často negativní dopad na cenu akcií, přičemž tento negativní dopad může přetrvávat dlouhou dobu. Investoři obvykle reagují negativně na zprávy o bezpečnostních incidentech, což vede k následnému prodeji akcií a poklesu celkové tržní hodnoty organizace [46].

4.2 Poškození reputace a ztráta důvěry

Útoky na softwarový dodavatelský řetězec mohou vážné poškodit reputaci organizace a narušit důvěru zainteresovaných stran.

4.2.1 Poškození důvěry klíčových investorů

- **Zákazníci:** Ztrácejí důvěru v bezpečnost produktů a schopnost organizace chránit jejich data, což může vést k jejich odchodu

- **Obchodní partneři:** Mohou mít obavy z rizika šíření útoku na jejich systémy a váhat s pokračováním zahájením spolupráce. Narušení důvěry může poškodit obchodní vztahy a vést ke ztrátě příležitostí. Partneři se mohou obávat kompromitace prostřednictvím sdílených sítí nebo dat [45].
- **Investoři:** Mohou zpochybnit stabilitu firmy a její schopnost řídit rizika [45]. To může vést k poklesu ceny akcií a ztížit získávání kapitálů. Incident je vnímán jako selhání řízení rizik.
- **Zaměstnanci:** Mohou být demotivováni, znepokojeni stabilitou firmy a obávat se o jistotu zaměstnání, což může vést k jejich odchodu. Dopad se pak následně šíří interně a ovlivňuje morálku [45].

4.2.2 Negativní publicita a mediální pokrytí

Negativní publicita po útoku poškozuje vnímání značky veřejností. Může vést k dlouhodobému „cejchování“ firmy jako nespolehlivé v oblasti bezpečnosti. Tato negativní pověst může přetrvávat dlouhé roky [45], ztěžovat získávání a udržení zákazníků a partnerů.

4.2.3 Potíže při získávání nových zákazníků

Poškozená reputace a negativní publicita přímo ztěžuje získání nových zákazníků a navazování nových obchodních partnerství. Potenciální partneři mohou také vnímat spolupráci jako příliš rizikovou.

4.3 Narušení provozu

Útoky na dodavatelský řetězec mohou způsobit závažná narušení běžného provozu organizace

4.3.1 Výpadky systémů a nedostupnost služeb

Jedním z bezprostředních dopadů takovýchto útoků může být přerušení kritických interních procesů, jako jsou výroba, logistika či administrativa. Zejména ransomwarové útoky mohou vést až k úplnému zastavení provozu organizace. Vzhledem k vysoké míře propojenosti moderních podnikových systémů může kompromitace jedné komponenty vyvolat kaskádový efekt a okamžitě ovlivnit klíčové činnosti.

Současně se mohou stát nedostupnými i externí služby a produkty určené zákazníkům, což vede k jejich nespokojenosti a potenciální ztrátě.

Provozní výpadky navíc mohou znemožnit plnění smluvních závazků a dohod vůči partnerům a zákazníkům. Nesplnění těchto závazků může vyústit ve finanční sankce a potenciálně i právní spory.

4.3.2 Zastavení nebo zpomalení vývojových procesů

Bezpečnostní incident v dodavatelském řetězci má často za následek nutnost zastavit nebo alespoň zpomalit probíhající softwarové vývojové procesy. Nevyhnutelnou součástí reakce na incident je provedení důkladného auditu a následné zabezpečení celého vývojového prostředí.

To zahrnuje revizi a posílení bezpečnosti CI/CD a dalších souvisejících nástrojů, aby se předešlo budoucím kompromitacím. Tento proces sanace a zabezpečení je typicky velmi náročný jak na čas, tak na zdroje. Narušení zavedených vývojových cyklů a odklon zdrojů na řešení incidentu může vést k nevyhnutelným zpožděním při uvádění nových produktů, funkcí nebo aktualizací na trh [47].

Takovéto zpoždění může negativně ovlivnit konkurenceschopnost organizace a v krajním případě vést i ke ztrátě tržního podílu.

4.3.3 Ztráta nebo poškození dat

Přímým následkem útoků na softwarový dodavatelský řetězec může být rovněž ztráta nebo poškození kritických dat, což vážně ohrožuje kontinuitu provozu. Snahy o obnovu dat ze záloh mohou být poměrně náročné.

Zvlášť závažné je riziko trvalé ztráty dat, které může nastat zejména v případech, kdy zálohovací mechaniky selžou, nejsou dostatečné, nebo byly taktéž kompromitovány útočníkem. Trvalá ztráta klíčových dat může mít pro postiženou organizaci až fatální existenční následky.

4.3.4 Přetížení a odklon zdrojů

Reakce na bezpečnostní incident spojený s dodavatelským řetězcem klade značnou neplánovanou zátěž na interní zdroje organizace, především na její IT a bezpečnostní týmy.

Tyto, často již tak vytížené, zdroje musí být nezřídka odkloněny od svých běžných provozních úkolů a plánovaných strategických projektů, aby se mohly plně věnovat identifikaci, omezení, vyšetřování a nápravě jak incidentu, tak jeho následků.

Toto vynucené přeměrování omezených zdrojů má pak za následek zpoždění v realizaci jiných důležitých obchodních a technologických iniciativ a celkově negativně ovlivňuje produktivitu a strategický rozvoj organizace [47].

4.4 Právní a regulační důsledky

Útoky na softwarový řetězec mohou mít také vážné právní a regulační následky.

4.4.1 Pokuty regulačních úřadů

Nedostatečná ochrana data systémů může vést k pokutám dle různých nařízení. Mezi ty patří například:

- **GDPR:** Pokuty až 4 % celosvětového ročního obratu, nebo 20 milionů EUR.
- **NIS2:** Pokuty až 10 milionů EUR nebo 2 % celosvětového ročního obratu
- **HIPAA (USA, zdravotnictví):** Pokuty až 50 000 amerických dolarů

Tabulka 1 Příklady regulačních pokut za narušení dat

Regulace	Potenciální pokuty
GDPR	20 milionů EUR nebo 4 % celosvětového obratu (vyšší částka)
NIS2	10 milionů EUR nebo 2 % celosvětového obratu (vyšší částka)
HIPAA	50 000 USD za porušení, maximálně však ~2 miliony USD ročně za opakované porušení

(zdroj: [48, 49])

4.4.2 Žaloby a soudní spory

Vedle pokut od regulačních orgánů čelí organizace postižené útokem také významnému riziku soudních sporů.

Častým jevem jsou hromadné žaloby podávané postiženými zákazníky, kteří požadují náhradu škody za únik osobních údajů nebo jiné újmy způsobené incidentem. Výše finančních vyrovnání v takových případech mohou dosahovat řádů milionů až stovek milionů amerických dolarů [50].

K podání žaloby mohou přistoupit také obchodní partneři, a to například z důvodů porušení smluvních závazků nebo škod vzniklých v důsledku rozšíření útoku do jejich vlastních systémů [44].

Riziko takových žalob je obzvláště relevantní, pokud smlouvy mezi partnery obsahují specifické doložky o kybernetické bezpečnosti a partner v důsledku incidentu utrpěl prokazatelné finanční ztráty.

4.4.3 Povinné audity a zvýšený dohled

V návaznosti na bezpečnostní incident mohou regulační úřady nařídit postižené organizace provedení povinných bezpečnostních auditů a implementaci specifických nápravných opatření. To s sebou nese nejen další náklady spojené se zajištěním dodržování předpisů, tzv. „compliance“, které navyšují celkové finanční dopady incidentu, ale často také vede k dlouhodobějšímu a intenzivnějšímu regulačnímu dohledu nad činností organizace v budoucnu.

4.4.4 Porušení smluvních závazků

Jak již bylo naznačeno (viz část 4.3.1), narušení provozu způsobené útokem může vést k neschopnosti organizace plnit své smluvní závazky vůči zákazníkům, nebo partnerům. Takové selhání může aktivovat smluvní sankce a v krajním případě vést k dalším právním sporům (viz 4.4.2).

4.5 Strategické a konkurenční dopady

Dopady útoků na softwarový dodavatelský řetězec přesahuje bezprostřední finanční, provozní a právní problémy a mohou zásadním způsobem ovlivnit dlouhodobou strategickou pozici a konkurenceschopnost napadené firmy.

4.5.1 Ztráta konkurenční výhody

Pokud útočníci během incidentu získají přístup k citlivému duševnímu vlastnictví, obchodním tajemstvím nebo jiným strategickým informacím, může organizace přijít o svou unikátní konkurenční výhodu na trhu.

Stejně tak, pokud je firma v důsledku útoku nucena stáhnout klíčové produkty z trhu, nebo výrazně zpozdit jejich uvedení či aktualizaci, poskytuje tím prostor konkurenci pro získání tržního náskoku.

4.5.2 Vliv na fúze a akvizice (M&A)

Závažný bezpečnostní incident může zkomplikovat, nebo dokonce zhatit probíhající jednání o fúzi či akvizici. Potenciální kupující, nebo partner může vnímat postiženou firmu jako příliš rizikovou, což může vést ke snížení její hodnoty, nebo k odstoupení od transakce. V posledních letech se proto hloubková kontrola kybernetické bezpečnosti (cybersecurity due diligence), včetně prověření odolnosti dodavatelského řetězce, stala kritickou součástí procesů M&A a navazování strategických partnerství [45].

4.5.3 Dlouhodobý dopad na inovace

Po úspěšném útoku jsou organizace často nuceny přesměrovat významnou část svých finančních i lidských zdrojů od inovativních projektů a vývoje nových produktů směrem k nápravě škod, posilování bezpečnostních opatření a obnově důvěry [45]. Tento odklon pozornosti a prostředků může vést ke zpomalení tempa inovací a budoucího rozvoje produktového portfolia, což v dlouhodobém horizontu negativně ovlivňuje schopnost firmy růst a udržet si konkurenceschopnost.

4.6 Faktory ovlivňující závažnost dopadů

Závažnost a rozsah výše popsaných dopadů nejsou konstantní, ale závisí na souhře několika klíčových faktorů:

- **Typ a sofistikovanost útoku:** Pokročilejší, cílenější a perzistentní útoky – například APT útoky, bývají obtížněji detekovatelné a mohou způsobit podstatně rozsáhlejší a déle trvající škody, než méně sofistikované, oportunistické útoky [45].
- **Rozsah kompromitace:** Přímý vliv na velikost dopadu má počet zasažených systémů a uživatelů, objem a citlivost kompromitovaných dat a počet postižených zákazníků či partnerů v dodavatelském řetězci [45]. Logicky platí, že čím širší je kompromitace, tím větší jsou výsledné ztráty.
- **Charakteristika zasažené organizace:** Velikost organizace a odvětví, ve kterém působí, hrají významnou roli. Útoky na provozovatele kritické infrastruktury mohou mít závažné celospolečenské důsledky. Organizace ve zdravotnictví čelí obzvláště vysokým nákladům kvůli extrémní citlivosti

pacientských dat. Naopak malé a střední podniky nemusí mít dostatek zdrojů na účinné zotavení po závažném incidentu [45].

- **Rychlost a efektivita reakce na incident:** Schopnost organizace rychle detekovat útok, omezit jeho šíření a efektivně reagovat může významně zmírnit celkový dopad [51]. Organizace s dobře připravenými a otestovanými plány reakce na incidenty (Incident Response Plans) jsou obecně lépe vybaveny pro zvládání krize. Rychlejší detekce a omezení prokazatelně snižují finanční náklady spojené s incidentem.
- **Transparentnost komunikace po incidentu:** Způsob, jakým organizace komunikuje o incidentu se zasaženými stranami (zákazníky, partnerny, regulátory, veřejností), má vliv na reputační škody. Otevřená, včasná a upřímná komunikace může pomoci zmírnit negativní dopady na reputaci a udržet důvěru [45], zatímco nedostatek transparentnosti nebo pokusy o zamlčení incidentu mohou škody naopak prohloubit.

4.7 Případová studie

Kybernetický útok na společnost SolarWinds v roce 2020 byl sofistikovaným útokem na dodavatelský řetězec, který měl rozsáhle globální dopady. Platforma SolarWinds Orion, široce používaný nástroj pro správu a monitoring IT infrastruktury, se stala vektorem útoku, který zasáhl tisíce organizací, včetně vládních agentur USA a velkých soukromých společností [52]. Za útokem stála skupina APT29 (známá také jako Nobelium nebo Cozy Bear), spojována s ruskou zahraniční zpravodajskou službou SVR [53, 54].

Útočníci nejprve kompromitovali vývojové prostředí společnosti SolarWinds. Následně vložili škodlivý kód, backdoor známý jak SUNBURST, přímo do legitimního souboru *SolarWinds.Orion.Core.BusinessLayer.dll* během procesu sestavování softwaru. Tato kompromitovaná komponenta byla poté digitálně podepsána certifikátem SolarWinds a distribuována zákazníkům jako běžná softwarová aktualizace mezi březnem a červnem 2020. Malware SUNBURST byl navržen tak, aby po instalaci zůstal po určitou dobu neaktivní a poté navázal komunikaci s řídicími servery využívající subdomény *avsvmcloud.com* pomocí protokolu, který napodoboval legitimní komunikaci platformy Orion, aby se vyhnul detekci [52, 55].

Útok bylo odhalen v prosinci 2020 společností FireEye, která jej objevila při vyšetřování vlastního bezpečnostního incidentu [54]. Následovala rychlá reakce – americká agentura CISA vydala nouzové nařízení pro federální úřady, aby okamžitě odpojily postižené systémy Orion [53], a společnost SolarWinds vydala bezpečnostní záplaty. Incident vedl k rozsáhlé spolupráci mezi soukromým sektorem a vládními agenturami při analýze a sdílení informací o této hrozbě.

Analýza incidentu odhalila nedostatky v bezpečnostních postupech společnosti SolarWinds, včetně nedostatečného zabezpečení vývojového prostředí a slabých hesel [54]. Případ zdůraznil kritickou potřebu robustního zabezpečení softwarového dodavatelského řetězce, důkladného prověřování dodavatelů, implementace principů „Zero Trust“ architektury a využívání Software Bill of Materials (SBOM) pro lepší transparentnost softwarových komponent. Útok také urychlil změny v regulačním prostředí, včetně zvýšeného tlaku na společnosti, aby transparentně informovaly o svých bezpečnostních postupech a incidentech [53].

4.8 Závěrečné zhodnocení

Útoky na softwarový dodavatelský řetězec představují kompletní a vysoce závažnou hrozbu pro organizace všech velikostí napříč různými odvětvími. Jejich nebezpečnost tkví ve zneužití důvěry a vzájemné propojenosti moderního softwarového ekosystému, kde kompromitace jediné komponenty může vést k exponenciálnímu šíření dopadů na nespočet koncových uživatelů a organizací. Tento typ útoku je obzvláště zákeřný, jelikož často obchází tradiční bezpečnostní opatření tím, že využívá legitimní kanály, jako jsou softwarové aktualizace z důvěryhodných zdrojů.

Dopady těchto incidentů jsou mnohostranné a hluboce zasahují do fungování postižených organizací. Sahají od okamžitých a často značných přímých finančních nákladů spojených s reakcí na incident, forenzní analýzou, obnovou systémů a případným výkupným, až po dlouhodobé nepřímé finanční ztráty způsobené narušením provozu, odchodem zákazníků, zvýšením pojistného a poklesem hodnoty akcií.

Provozní narušení, včetně výpadků systémů, zastavení vývojových procesů a potenciální ztráty kritických dat, mohou paralyzovat činnost firmy a znemožnit plnění smluvních závazků. Právní a regulační důsledky, zahrnují vysoké pokuty od úřadů, nákladné soudní spory a zpřísněný dohled, dále zvyšují celkovou zátěž incidentu. V neposlední řadě

mohou strategické dopady, jako je ztráta konkurenční výhody či negativní vliv na fúze a akvizice, zásadně ovlivnit dlouhodobou pozici a životaschopnost organizace na trhu.

Případová studie útoku na SolarWinds slouží jako názorná ukázka devastujícího potenciálu těchto hrozeb a zároveň zdůrazňuje kritickou potřebu robustních bezpečnostních opatření. Závažnost dopadů sice závisí na řadě faktorů, včetně sofistikovanosti útoku a připravenost organizace, avšak podcenění rizika si nemůže dovolit žádný subjekt spoléhající na software. Zabezpečení softwarového dodavatelského řetězce, zahrnující důkladné prověřování dodavatelů, implementaci principů jako „Zero Trust“, využívání SBOM a posílení bezpečnosti vývojových procesů, se tak stává naprosto nezbytnou součástí komplexní strategie kybernetické bezpečnosti a řízení rizik.

II. PRAKTICKÁ ČÁST

5 STRATEGIE PRO ZMÍRNĚNÍ RIZIK ÚTOKŮ NA DODAVATELSKÝ ŘETĚZEC

Předchozí kapitoly analyzovaly problematiku útoků na softwarový dodavatelský řetězec, identifikovaly klíčové typy hrozeb jako dependency confusion či kompromitace systémů pro sestavení a popsaly jejich potenciálně devastující dopady, jak ilustruje například případ SolarWinds. Ukázalo se, že moderní softwarový vývoj, charakteristický vysokou mírou závislosti na externích komponentách a službách, představuje komplexní ekosystém s mnoha vektory útoku. Tato kapitola se věnuje návrhu strategie pro zmírnění identifikovaných rizik, která kombinuje technická, procesní a organizační opatření k posílení odolnosti celého dodavatelského řetězce.

5.1 Principy a cíle strategie zmírnění rizik

Účinná strategie pro zmírnění rizik spojených s útoky na softwarový dodavatelský řetězec by měla vycházet z několika základních bezpečnostních principů a sledovat jasně definované cíle. Mezi klíčové principy patří:

5.1.1 Zero Trust

Pro aplikování principu Zero Trust je nutné nevěřit žádné společnosti, závislosti, balíčku a podobně. Je potřeba důkladně ověřit původ bez ohledu od koho daná závislost pochází. K tomu můžou sloužit různé open-source či komerční nástroje, ale existuje také vestavěná funkcionality v operačním systému.

```
~\Downloads\test (0.062s)
gpg --print-md SHA1 gpg4win-4.4.0.exe
gpg4win-4.4.0.exe: 5C5E 06F9 F36D 816B A148 47D8 1312 7C95 1083 6394

~\Downloads\test (0.072s)
gpg --print-md SHA256 gpg4win-4.4.0.exe
gpg4win-4.4.0.exe: 76567385 4C150360 2B09C97B FA6C72B5 34E24141 85FB2F23
A0CE19CF 8CECD891

~\Downloads\test (0.092s)
Get-FileHash gpg4win-4.4.0.exe -Algorithm SHA1

Algorithm      Hash
-----
SHA1           5C5E06F9F36D816BA14847D813127C9510836394

~\Downloads\test (0.061s)
Get-FileHash gpg4win-4.4.0.exe -Algorithm SHA256

Algorithm      Hash
-----
SHA256         765673854C1503602B09C97BFA6C72B534E2414185FB2F23A0CE19CF8CECD891
```

Obrázek 1 Výstup z příkazové řádky

(zdroj: vlastní)

5.1.2 Defense in Depth

Ochrana by neměla spoléhat pouze na jedno opatření, naopak je potřeba implementovat více vrstev. Pro ukázkou jsem instalaci dvou npm balíčků – **express@5.0.0** a **bash-parser@0.5.0**.

1. **Zajištění integrity a původu:** Instalace pomocí *npm ci* zajistí, že příkaz selže, pokud se neshodují balíčky mezi soubory **package.json** a **package-lock.json**.

```
npm ci
npm error code EUSAGE
npm error
npm error `npm ci` can only install packages when your package.json and package-lock.json or
npm-shrinkwrap.json are in sync. Please update your lock file with `npm install` before con
tinuing.
npm error
npm error Invalid: lock file's express@4.7.1 does not satisfy express@5.0.0
npm error Invalid: lock file's accepts@1.0.7 does not satisfy accepts@2.0.0
npm error Missing: body-parser@2.2.0 from lock file
npm error Missing: content-disposition@1.0.0 from lock file
npm error Missing: content-type@1.0.5 from lock file
npm error Invalid: lock file's cookie@0.1.2 does not satisfy cookie@0.6.0
npm error Invalid: lock file's cookie-signature@1.0.4 does not satisfy cookie-signature@1.2.
2
```

Obrázek 2 Výstup příkazu npm ci

(zdroj: vlastní)

2. **Skenování známých zranitelností:** Použitím příkazu `npm audit` je možné zkontrolovat, zda se mezi balíčky nenachází nějaký se známou zranitelností. Pomocí parametru `--audit-level` je možné nastavit, při jaké závažnosti příkaz selže.

```
npm audit --audit-level=high
# npm audit report

cookie <0.7.0
cookie accepts cookie name, path, and domain with out of bounds characters - https://github.
com/advisories/GHSA-pxg6-pf52-xh8x
fix available via `npm audit fix --force`
Will install express@5.1.0, which is outside the stated dependency range
node_modules(cookie)
  express 3.0.0-alpha1 - 4.21.0 || 5.0.0-alpha.1 - 5.0.0
  Depends on vulnerable versions of cookie
  node_modules/express

2 low severity vulnerabilities

To address all issues, run:
npm audit fix --force
```

Obrázek 3 npm audit

(zdroj: vlastní)

3. **Kontrola licencí v použitých závislostech:** Je možné narazit nejen na bezpečnostní problémy, ale také i na licenční. Z toho důvodů je velmi důležité kontrolovat také licence, pod kterými jsou závislosti vydány. Níže je ukázka příkazu, který selže, pokud se mezi licencemi objeví například licence GPL.

```
npx license-checker --failOn "AGPL*;GPL*;LGPL*"
├─ accepts@2.0.0
│  ├── licenses: MIT
│  ├── repository: https://github.com/jshttp/accepts
│  └── path: C:\Users\zamec\Documents\UTB\diplomova-prace\prakticka-cast\simple-server\node_modules\accepts
├─ arity-n@1.0.4
│  ├── licenses: MIT
│  ├── repository: https://github.com/stoeffel/arityN
│  ├── publisher: stoeffel
│  └── path: C:\Users\zamec\Documents\UTB\diplomova-prace\prakticka-cast\simple-server\node_modules\arity-n
└─ licenseFile: C:\Users\zamec\Documents\UTB\diplomova-prace\prakticka-cast\simple-server\node_modules\accepts\LICENSE
└─ licenseFile: C:\Users\zamec\Documents\UTB\diplomova-prace\prakticka-cast\simple-server\node_modules\arity-n\LICENSE
```

Obrázek 4 npx license-checker

(zdroj: vlastní)

5.1.3 Shift-Left Security

Shift-Left Security integruje bezpečnostní kontrolu co nejdříve do životního cyklu. Například, je možné kontrolovat zranitelné balíčky na produkčním serveru. Kontrola sice proběhne, ale zranitelnosti jsou již v produkci, což je špatně. S použitím Shift-Left Security se kontrola posouvá do dřívější části vývoje, což může být například v CI/CD pipelinech. Ale je možné jít ještě dále, a kontrolu provádět již při commitu kódu, čemuž se věnuje ukázka níže, která využívá nástroj **husky** a **npm audit** pro kontrolu zranitelností.

1. **Husky**: Nástroj husky je nakonfigurován tak, aby při commitu využil **npm audit** pro kontrolu známých zranitelností.

```
1  echo "Running pre-commit checks..."
2
3  npm audit --audit-level=moderate
4
5  audit_exit_code=$?
6
7  if [ $audit_exit_code -ne 0 ]; then
8      echo "npm audit found vulnerabilities. Commit REJECTED!"
9      exit 1
10 fi
11
12 echo "npm audit passed."
13 exit 0
14
```

Obrázek 5 Konfigurace Pre-commitu

(zdroj: vlastní)

2. **Vývojář přidá nový balíček:** Pro tuto ukázkou je zvolen balíček **express** na verzi **4.7.2**, která obsahuje známé zranitelnosti.

```
11  "author": "Petr Zamecnik",
12  "license": "MIT",
13  "dependencies": {
14    "bash-parser": "0.5.0",
15    "express": "4.7.2"
16  },
17  "devDependencies": {
18    "husky": "^9.1.7"
19  }
```

Obrázek 6 Ukázka package.json

(zdroj: vlastní)

3. **Přidání změn do repozitáře:** Následně se vývojář pokusí přidat změny do repozitáře. Jak je vidět níže, již samotný commit se nezdaří. Díky vhodné konfiguraci a využití Shift-Left Security, se zranitelnosti nedostanou ani do repozitáře.

```
git commit -m "add express package"
Running pre-commit checks...
# npm audit report

cookie <0.7.0
cookie accepts cookie name, path, and domain with out of bounds characters - https://github.com/advisories/GHSA-pxg6-pf52-xh8x
fix available via `npm audit fix --force`
Will install express@4.21.2, which is outside the stated dependency range
node_modules/cookie
```

Obrázek 7 Selhání git commit 1/2

(zdroj: vlastní)

```
debug <=2.6.8
Severity: high
debug Inefficient Regular Expression Complexity vulnerability - https://github.com/advisories/GHSA-9vbw-cc9w-f27h
Regular Expression Denial of Service in debug - https://github.com/advisories/GHSA-gxpj-cx7g-858c
Depends on vulnerable versions of ms
fix available via `npm audit fix --force`
Will install express@4.21.2, which is outside the stated dependency range
node_modules/debug
```

Obrázek 8 Selhání git commit 2/2

(zdroj: vlastní)

5.1.4 Transparentnost a viditelnost

Organizace by měla mít přehled o všech komponentách a závislostech. Včetně tranzitivních, jelikož i mezi nimi se může skrývat zranitelnost, nebo licence která naprosto nesouhlasí s firemní politikou. Takovému přehledu se říká Software Bill of Materials – SBOM, a je možné jej generovat pomocí nástrojů jako je Trivy, nebo CycloneDX. V ukázce níže je použit právě nástroj Trivy.

```
trivy fs --format cyclonedx --include-dev-deps --scanners vuln --output sbom.json .
2025-05-01T13:35:33+02:00      INFO      [vuln] Vulnerability scanning is enabled
2025-05-01T13:35:33+02:00      INFO      Number of language-specific files          num=1
2025-05-01T13:35:33+02:00      INFO      [npm] Detecting vulnerabilities...
```

Obrázek 9 Generování SBOM za pomoci Trivy

(zdroj: vlastní)

Výstupem tohoto příkazu bude soubor **sbom.json**, který obsahuje seznam všech nalezených komponent a detailní informace o jednotlivých komponentách. Například takto vypadá balíček **express@5.0.0**:

```
{
  "bom-ref": "pkg:npm/express@5.0.0",
  "type": "library",
  "name": "express",
  "version": "5.0.0",
  "licenses": [
    {
      "license": {
        "name": "MIT"
      }
    }
  ],
  "purl": "pkg:npm/express@5.0.0",
  "properties": [
    {
      "name": "aquasecurity:trivy:PkgID",
      "value": "express@5.0.0"
    },
    {
      "name": "aquasecurity:trivy:PkgType",
      "value": "npm"
    }
  ]
},
```

Obrázek 10 Ukázka json souboru zobrazující balíček express

(zdroj: vlastní)

5.1.5 Automatizace

Vzhledem k tomu, jak je moderní vývoj komplexní, a jak rychle probíhají změny, automatizace těchto procesů je téměř nezbytná. Níže je jednoduchá ukázka automatizace v CI/CD pipeline.

1. Nastavení pipeline

```
1  name: Automated Security Pipeline
2
3  on:
4    push:
5      branches:
6        - main
7    pull_request:
8      branches:
9        - main
10   workflow_dispatch:
11
12
13  jobs:
14    security-checks:
15      name: Run Automated Security Checks
16      runs-on: ubuntu-22.04
17      permissions:
18        actions: read
19        contents: read
20        security-events: write
21
22      steps:
23        - name: Checkout repository
24          uses: actions/checkout@v4
25
26        - name: Set up Node.js
27          uses: actions/setup-node@v4
28          with:
29            node-version: "20.x"
30            cache: "npm"
31
```

Obrázek 11 Ukázka GitHub Actions pro automatizovanou bezpečnostní kontrolu

(zdroj: vlastní)

2. Instalace Node.js, závislostí a kontrola zranitelností

```
31
32     # Automatizace 1: Instalace Node.js, závislostí a kontrola zranitelností
33     - name: Install dependencies
34       run: npm ci
35
36     - name: Scan Node.js dependencies for security vulnerabilities (high)
37       run: npm audit --audit-level=high
38
39
40
41
```

Obrázek 12 Instalace Node.js, závislostí a kontrola zranitelností

(zdroj: vlastní)

3. Generování SBOM

```
40     # Automatizace 2: Generování SBOM pomocí Trivy
41     - name: Generate SBOM
42       uses: aquasecurity/trivy-action@master
43       with:
44         scan-type: "fs"
45         scan-ref: "."
46         format: "cyclonedx"
47         output: "sbom.cyclonedx.json"
48         exit-code: "0"
49       env:
50         TRIVY_INCLUDE_DEV_DEPS: "true"
51
```

Obrázek 13 Ukázka generování SBOM za pomoci Trivy

(zdroj: vlastní)

4. Skenování zranitelností za pomoci Trivy

```
50      # Automatizace 3: Skenování zranitelností pomocí trivy
51      - name: Scan filesystem for vulnerabilities
52      uses: aquasecurity/trivy-action@master
53      with:
54        scan-type: "fs"
55        scan-ref: "."
56        format: "table"
57        severity: "HIGH,CRITICAL"
58        exit-code: "1"
59
```

Obrázek 14 Skenování zranitelností pomocí Trivy

(zdroj: vlastní)

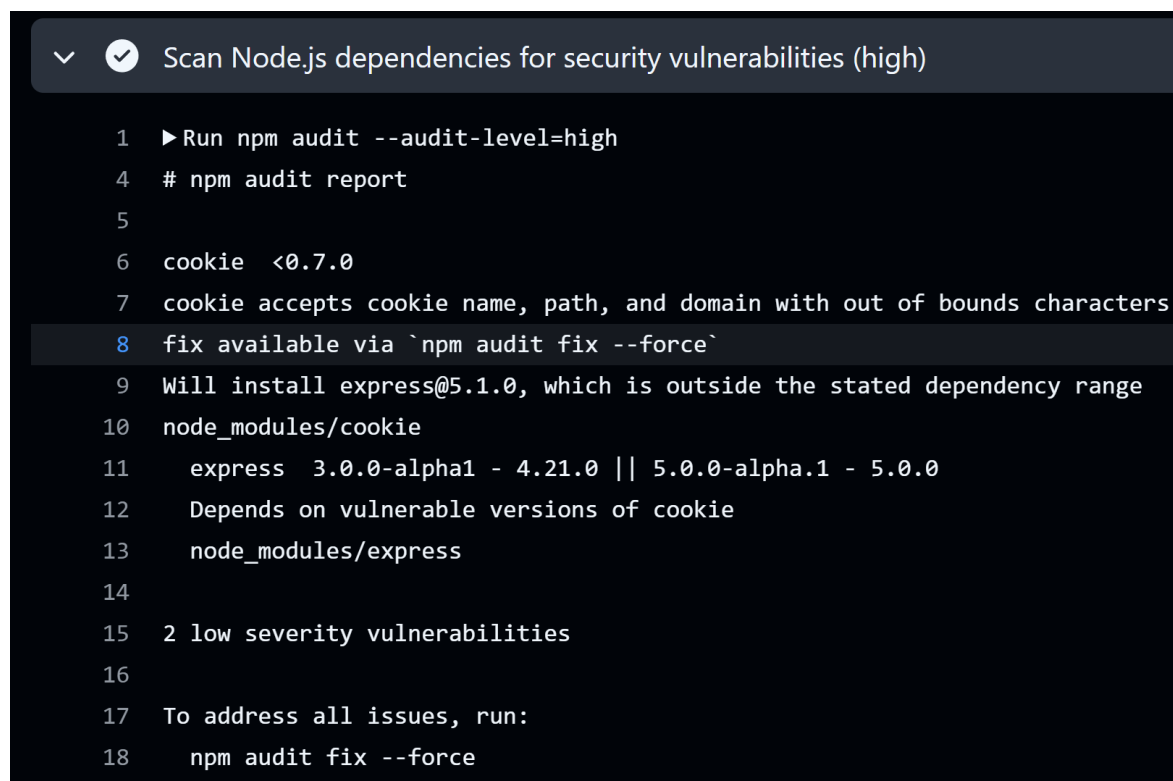
5. Nahrání vygenerovaného SBOM jako artefaktu

```
60      # Automatizace 4: Nahrání SBOM jako artefaktu
61      - name: Upload SBOM artifact
62      uses: actions/upload-artifact@v4
63      with:
64        name: sbom report
65        path: sbom.cyclonedx.json
66        retention-days: 7
67
```

Obrázek 15 Nahrání SBOM souboru jako artefaktu

(zdroj: vlastní)

První krok automatizace našel dvě zranitelnosti s malou závažností, tento krok tedy prošel bez chyb.

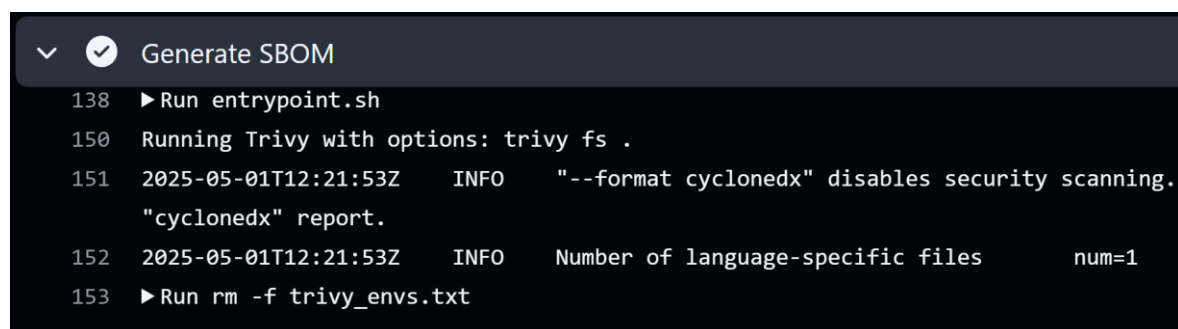


```
1  ▶ Run npm audit --audit-level=high
4  # npm audit report
5
6  cookie <0.7.0
7  cookie accepts cookie name, path, and domain with out of bounds characters
8  fix available via `npm audit fix --force`
9  Will install express@5.1.0, which is outside the stated dependency range
10 node_modules/cookie
11   express 3.0.0-alpha1 - 4.21.0 || 5.0.0-alpha.1 - 5.0.0
12   Depends on vulnerable versions of cookie
13   node_modules/express
14
15  2 low severity vulnerabilities
16
17  To address all issues, run:
18    npm audit fix --force
```

Obrázek 16 Ukázka skenu z GitHub Actions pro kontrolu zranitelností

(zdroj: vlastní)

Druhý krok automatizace úspěšně vygeneroval SBOM.



```
138 ▶ Run entrypoint.sh
150 Running Trivy with options: trivy fs .
151 2025-05-01T12:21:53Z INFO "--format cyclonedx" disables security scanning.
    "cyclonedx" report.
152 2025-05-01T12:21:53Z INFO Number of language-specific files num=1
153 ▶ Run rm -f trivy_envs.txt
```

Obrázek 17 Ukázka z GitHub Actions při generování SBOM

(zdroj: vlastní)

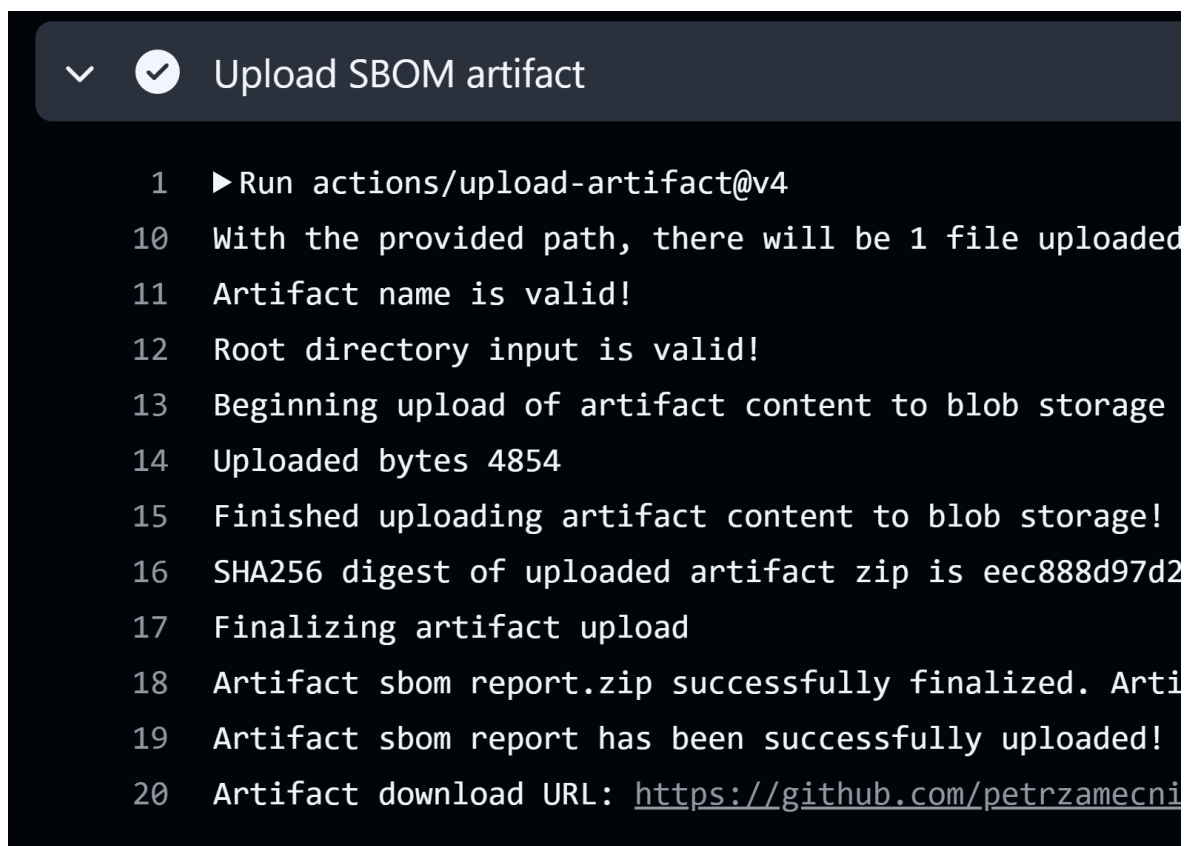
Třetí krok zkontroloval zranitelností a žádné nenašel.

```
139  Scan filesystem for vulnerabilities
140
141  Report Summary
142
143  | Target | Type | Vulnerabilities | Secrets |
144  |-----|-----|-----|-----|
145  | package-lock.json | npm | 0 | - |
146
147  Legend:
148  - '-': Not scanned
149  - '0': Clean (no security findings detected)
150  ▶ Run rm -f trivy_envs.txt
```

Obrázek 18 Ukázka skenu při kontrole zranitelností

(zdroj: vlastní)

Čtvrtý krok automatizace nakonec nahrál soubor **sbom.cyclonedx.json**, který byl vygenerován v druhém kroku, jako artefakt do GitHubu

The image shows a screenshot of a GitHub Actions workflow run. At the top, there is a header bar with a downward arrow, a checkmark icon, and the text 'Upload SBOM artifact'. Below this, the workflow steps are listed with line numbers. Step 1 is 'Run actions/upload-artifact@v4'. Subsequent lines show the execution output: 'With the provided path, there will be 1 file uploaded', 'Artifact name is valid!', 'Root directory input is valid!', 'Beginning upload of artifact content to blob storage', 'Uploaded bytes 4854', 'Finished uploading artifact content to blob storage!', 'SHA256 digest of uploaded artifact zip is eec88d97d2', 'Finalizing artifact upload', 'Artifact sbom report.zip successfully finalized. Arti', 'Artifact sbom report has been successfully uploaded!', and 'Artifact download URL: https://github.com/petrzamecni'.

```
1 ▶Run actions/upload-artifact@v4
10 With the provided path, there will be 1 file uploaded
11 Artifact name is valid!
12 Root directory input is valid!
13 Beginning upload of artifact content to blob storage
14 Uploaded bytes 4854
15 Finished uploading artifact content to blob storage!
16 SHA256 digest of uploaded artifact zip is eec88d97d2
17 Finalizing artifact upload
18 Artifact sbom report.zip successfully finalized. Arti
19 Artifact sbom report has been successfully uploaded!
20 Artifact download URL: https://github.com/petrzamecni
```

Obrázek 19 Nahrání SBOM artefaktu

(zdroj: vlastní)

5.1.6 Hlavní cíle

Navrhovaná strategie, postavená na principech jako Zero Trust, Defense in Depth, Shift-Left Security, transparentnosti a automatizuje, sleduje několik klíčových cílů.

- Zvýšení odolnosti
- Zajištění integrity softwaru
- Budování a udržení důvěry
- Rychlá detekce zranitelností a efektivní reakce

5.2 Zabezpečení vývojového životního cyklu

Základem robustní strategie proti útokům na dodavatelský řetězec je implementace principů bezpečného vývojového životního cyklu (Secure Software Development Lifecycle – Secure

SDLC). Namísto reaktivního řešení bezpečnostních problémů až po jejich objevení, Secure SDLC prosazuje proaktivní integraci bezpečnostních praktik a kontrol do každé fáze vývojového procesu. Tím se nejen snižuje riziko zavedení zranitelností do vlastního kódu, ale také se zvyšuje odolnost vůči hrozbám pocházejícím z externích zdrojů a závislostí [3].

5.2.1 Fáze plánování

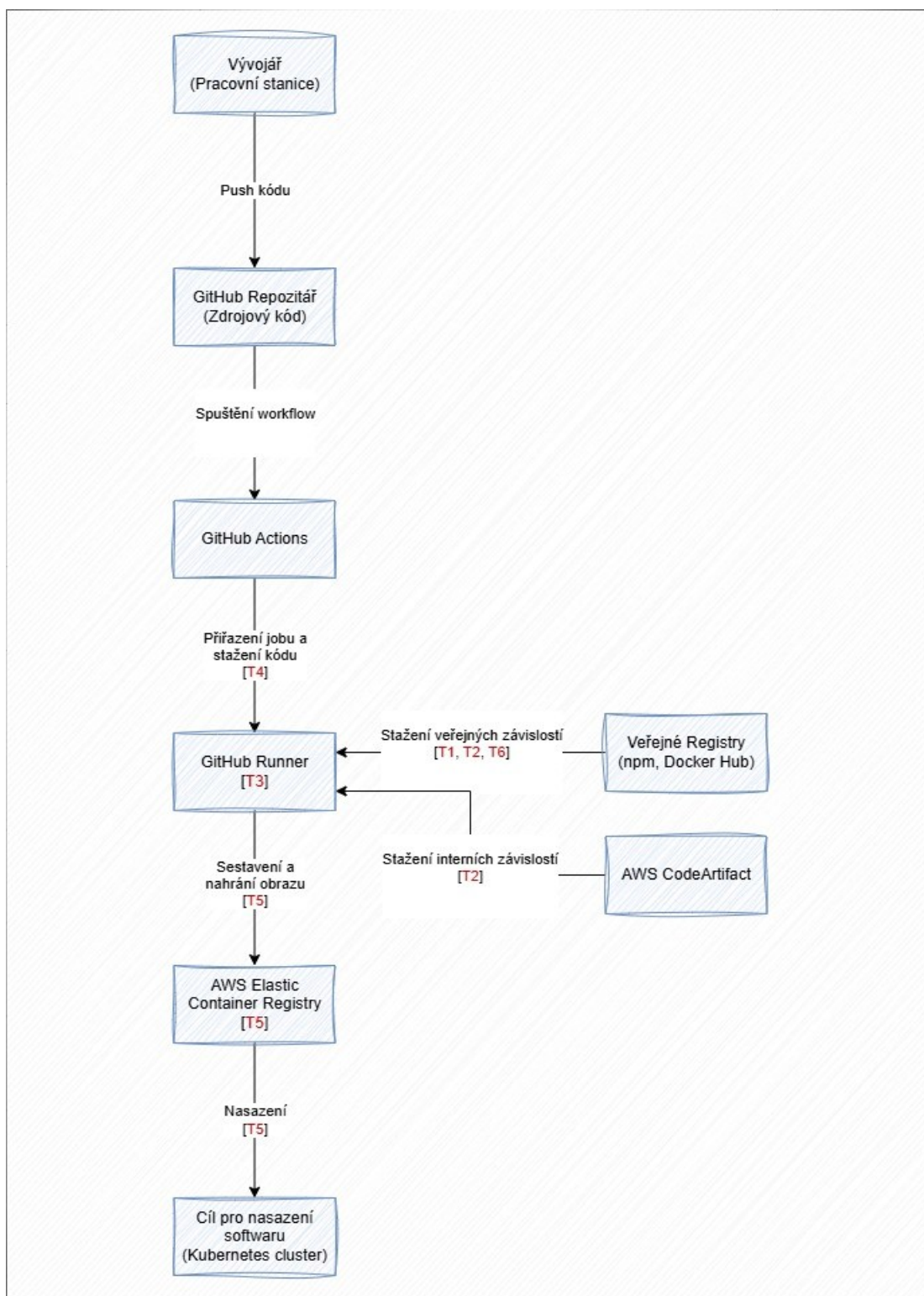
Již při definování funkčních požadavků je nutné specifikovat i bezpečnostní požadavky. To zahrnuje například požadavky na ověřování původu závislostí, bezpečné ukládání citlivých dat nebo odolnost proti konkrétním typům útoků relevantních pro danou aplikaci a její dodavatelský řetězec

5.2.2 Fáze návrhu

V této fázi je klíčové provádět tzv. **modelování hrozeb**. Jedná se o strukturovaný proces identifikace potenciálních hrozeb, zranitelností a definování protiopatření již na úrovni architektury a návrhu systému. Se zaměřením na dodavatelský řetězec by modelování hrozeb mělo analyzovat rizika spojená s:

- Použitím externích knihoven a framework
- Integraci s externími službami a API
- Použitím nástrojů pro správu balíčků
- Procesem sestavení a CI/CD pipeline
- Distribucí a aktualizací softwaru

Zjednodušený příklad modelování hrozeb pro webovou aplikaci používající Node.js, npm a GitHub CI/CD:



Obrázek 20 Zjednodušený model hrozeb pro webovou aplikaci

(zdroj: vlastní)

Tabulka 2 Modelování hrozeb

ID hrozby	Popis Hrozby (scénář)	Kategorie STRIDE	Cílová komponenta / datový tok	Potenciální dopad	Možná protipatření
T1	Stažení škodlivé závislosti (Dependency Confusion / Typosquatting): GitHub Runner stáhne balíček z veřejného repozitáře se stejným názvem jako má interní balíček, ale obsahuje škodlivý kód.	Spoofing / Tampering	GitHub Runner – Registry balíčků	Spuštění škodlivého kódu na runneru (RCE), krádež přihlašovacích údajů, kompromitace procesu sestavení nebo infikace finálního obrazu.	Použití AWS CodeArtifact jako primárního zdroje a proxy pro veřejné registry. Správná konfigurace správce balíčků. Využití scopes @organizace/balicek, rezervace názvů ve veřejném registru.
T2	Použití kompromitované závislosti (interní i veřejné): Runner stáhne legitimní balíček, který byl kompromitován.	Tampering	GitHub Runner – Veřejné registry GitHub Runner – AWS CodeArtifact	RCE na runneru, krádež dat, infikování artefaktu, šíření škodlivého kódu k zákazníkům.	SCA skenování, SBOM, prověřování závislostí, princip minimálního oprávnění pro runner.
T3	Kompromitace GitHub Runnera: Útočník získá kontrolu nad prostředím runnera.	Tampering / Elevation of Privilege / Information Disclosure	GitHub Runner	Manipulace se zdrojovým kódem, vložení backdooru, krádež citlivých informací.	Použití efemérních runerrů, izolace sítě, skenování obrazu runnera, princip nejnižších oprávnění, rotace klíčů.
T4	Manipulace s Workflow/Build skriptem: Útočník upraví .github/workflows nebo skripty pro sestavení.	Tampering	GitHub Actions / GitHub Runner	Spuštění libovolných příkazů na runneru, exfiltrace kódu, proměnných, modifikace artefaktu, vypnutí bezpečnostních kroků.	Ochrana větví v GitHubu, audit logy, omezení oprávnění, SAST pro GitHub workflow soubory.

T5	Manipulace s Docker obrazem: Útočník modifikuje obraz během sestavování, nahrávání do AWS ECR, nebo přímo v ECR.	Tampering	GitHub Runner – AWS ECR – Cíl pro deploy	Nasazení škodlivého kódu do produkce (kubernetes), kompromitace clusteru, krádež dat zákazníků.	Digitální podepisování obrazů a ověření podpisu před nasazením, striktní IAM politiky pro ECR, použití neměnných tagů v ECR, skenování obrazů v ECR.
T6	Zranitelnost v externí GitHub Action nebo základním obraze: Použitá externí GitHub Action (např. actions/checkout@v3) nebo základní Docker obraz obsahují zranitelnost.	Tampering / Elevation of Privilege	GitHub Runner	Eskalace oprávnění na runneru, únik citlivých dat.	Pinning verzí GitHub Actions na konkrétní commit hash (nejen tag), pravidelná revize, skenování obrazů, používání ověřených GitHub Actions

(zdroj: vlastní)

5.2.3 Fáze implementace

Zásadní je dodržování bezpečnostních standardů (např. OWASM Secure Coding Practices, CERT Secure Coding Standards), které se snaží minimalizovat běžné chyby vedoucí ke zranitelnostem (SQL Injection, XSS, chyby v řízení přístupu). Zvláštní pozornost musí být věnována bezpečnému používání externích závislostí – například důsledná validace a ošetření dat přicházejících z externích knihoven.

Také pravidelné revize kódu (Code Reviews), prováděné kolegy, jsou kritickým kontrolním bodem. Tyto revize by se měly explicitně zaměřovat i na nově přidané, nebo aktualizované závislosti, jejich konfiguraci a způsob použití v kódu.

5.2.4 Fáze testování

Kromě funkčního testování je nezbytné provádět i bezpečnostní testování. Zde hrají klíčovou roli nástroje pro statickou analýzu bezpečnosti kódu (SAST – Static Application Security Testing) a dynamickou analýzu bezpečnosti aplikací (DAST – Dynamic Application Security Testing).

- SAST nástroje analyzují zdrojový kód, bytecode nebo binární kód bez jeho spuštění a hledají vzorce kódu, které naznačují potencionální bezpečnostní zranitelnosti (např. použití nebezpečných funkcí, chybějící validace vstupu). Mohou analyzovat i kód používaných závislostí, pokud je jejich zdrojový kód dostupný a především způsob, jakým jsou tyto závislosti v aplikaci volány.
- DAST nástroje testují běžící aplikaci zvenčí, posílají různé typy vstupů a sledují reakce aplikace, aby odhalily zranitelnost jako XSS, SQL Injection a případně problémy s konfigurací serveru. Testují tedy i chování vyplývající z integrovaných závislostí.

Praktická ukázka integrace SAST a skenování citlivých informací:

Pro praktickou ukázku využívám nebezpečný javascriptový kód, který obsahuje citlivé informace jako API klíče, a potenciální bezpečnostní rizika. Mezi použité nástroje patří GitHub Actions, Gitleaks a ESLint.

Citlivé informace v kódu

```
const api_key : string = "zYxWvUtSrQpOnMLKjIhGfEdCbA987asd6543210ggs"  
const SECRET_ACCESS_TOKEN : string = "zYxWvUtSrQpOnMLKjIhGfEdCbA9876543210.=foo"  
const super_secret : string = "abc132"
```

Obrázek 21 Ukázka citlivých informací v kódu

(zdroj: vlastní)

Citlivé informace v souboru .env

```
API_KEY=zYxWvUtSrQpOnMLKjIhGfEdCbA987asd6543210ggs  
SECRET_ACCESS_TOKEN=zYxWvUtSrQpOnMLKjIhGfEdCbA9876543210.=foo  
SUPER_SECRET=abc132
```

Obrázek 22 Ukázka citlivých informací v souboru .env

(zdroj: vlastní)

Potenciálně nebezpečné použití *eval*

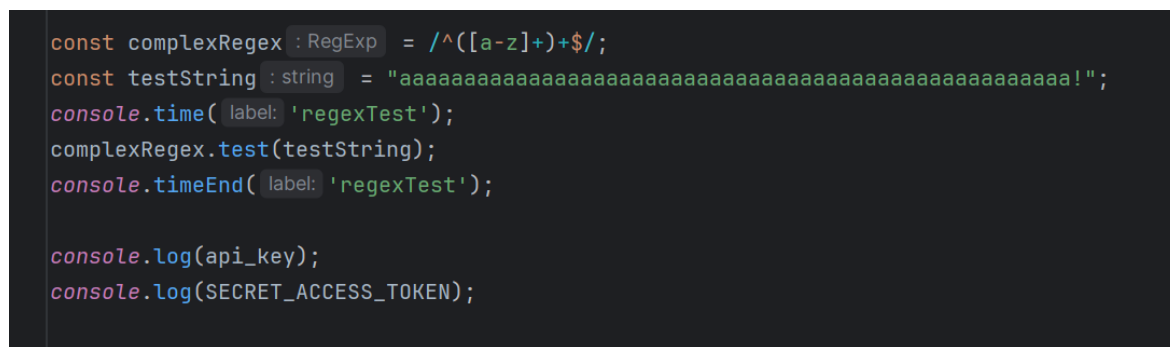


```
function executeDynamicCode(codeString) : void {  
  console.log("Spouštím kód:", codeString);  
  eval(codeString);  
}
```

Obrázek 23 Ukázka rizikového použití eval

(zdroj: vlastní)

Potenciálně nebezpečný regex (ReDoS)



```
const complexRegex : RegExp = /^[a-z]+)$/;  
const testString : string = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!";  
console.time( label: 'regexTest');  
complexRegex.test(testString);  
console.timeEnd( label: 'regexTest');  
  
console.log(api_key);  
console.log(SECRET_ACCESS_TOKEN);
```

Obrázek 24 Ukázka rizikového použití regex

(zdroj: vlastní)

GitHub Action použitý pro skenování problémů v kódu

```
1      name: Security Scan
2
3      on:
4        push:
5          branches: [ main, master ]
6        pull_request:
7          branches: [ main, master ]
8        workflow_dispatch:
9
```

Obrázek 25 GitHub Action pro skenování kódu 1/2

(zdroj: vlastní)

```
11     jobs:
12       scan:
13         runs-on: ubuntu-latest
14         steps:
15           - name: Checkout code
16             uses: actions/checkout@v3
17             with:
18               fetch-depth: 0
19
20           - name: Setup Node.js
21             uses: actions/setup-node@v3
22             with:
23               node-version: "18"
24
25           - name: Install dependencies (incl. ESLint)
26             run: |
27               npm install express
28               npm install --save-dev eslint eslint-plugin-security
29
```

Obrázek 26 GitHub Action pro skenování kódu 2/2

(zdroj: vlastní)

GitHub Action pro skenování zranitelných závislostí

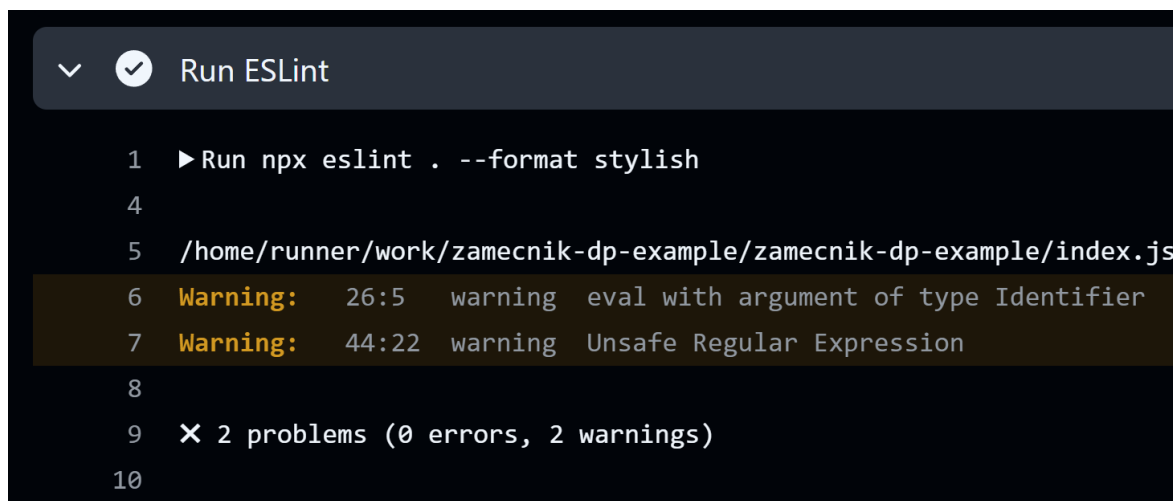
```
name: Trivy Dependency Scan
on: [push]
jobs:
  build:
    name: Build
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: Run Trivy filesystem scanner
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: 'fs'
          scan-ref: '.'
          ignore-unfixed: false
          format: 'table'
          exit-code: '1'
          severity: 'CRITICAL,HIGH,MEDIUM'
```

Obrázek 27 GitHub Action pro skenování zranitelných závislostí

(zdroj: vlastní)

Výsledek ESLint skenu:

- Úspěšně detekována nebezpečné použití funkce *eval*
- Úspěšně detekován potenciálně nebezpečný *regex*



```
1 ▶Run npx eslint . --format stylish
4
5 /home/runner/work/zamecnik-dp-example/zamecnik-dp-example/index.js
6 Warning: 26:5 warning eval with argument of type Identifier
7 Warning: 44:22 warning Unsafe Regular Expression
8
9 ✖ 2 problems (0 errors, 2 warnings)
10
```

Obrázek 28 Výsledek ESLint skenu

(zdroj: vlastní)

Výsledek Trivy skenu:



Target	Type	Vulnerabilities	Secrets
package-lock.json	npm	2	-

Legend:

- '-': Not scanned
- '0': Clean (no security findings detected)

Obrázek 29 Výsledek Trivy skenu 1/3

(zdroj: vlastní)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version
jquery	CVE-2020-11022	MEDIUM	fixed	3.4.1	3.5.0
	CVE-2020-11023				

Obrázek 30 Výsledek Trivy skenu 2/3

(zdroj: vlastní)

Title
jquery: Cross-site scripting due to improper injQuery.htmlPrefilter method https://avd.aquasec.com/nvd/cve-2020-11022
jquery: Untrusted code execution via <option> tag in HTML passed to DOM... https://avd.aquasec.com/nvd/cve-2020-11023

Obrázek 31 Výsledek Trivy skenu 3/3

(zdroj: vlastní)

Výsledek Gitleaks skenu:

```
11 Finding:      API_KEY=REDACTED
12 Secret:      REDACTED
13 RuleID:      generic-api-key
14 Entropy:      5.249460
15 File:        .env
16 Line:        3
17 Commit:      b367338d9313b0996bdf92cd3824b4d52dfc0f36
18 Author:      Petr Zamecnik
19 Email:       pzamecnik@pm.me
20 Date:        2025-04-13T01:04:22Z
21 Fingerprint: b367338d9313b0996bdf92cd3824b4d52dfc0f36:
22 Link:        https://github.com/petrzamecnik/zamecnik-
```

Obrázek 32 Výsledek Gitleaks skenu 1/2

(zdroj: vlastní)

```
24 Finding:      const api_key = "REDACTED"
25 Secret:      REDACTED
26 RuleID:      generic-api-key
27 Entropy:      5.249460
28 File:        index.js
29 Line:        7
30 Commit:      b367338d9313b0996bdf92cd3824b4d52dfc0f36
31 Author:      Petr Zamecnik
32 Email:       pzamecnik@pm.me
33 Date:        2025-04-13T01:04:22Z
34 Fingerprint: b367338d9313b0996bdf92cd3824b4d52dfc0f36:
35 Link:        https://github.com/petrzamecnik/zamecnik-
```

Obrázek 33 Výsledek Gitleaks skenu 2/2

(zdroj: vlastní)

- Úspěšně detekována citlivá informace *API_KEY*
- Úspěšně detekována citlivá informace *SECRET_ACCESS_TOKEN*
- Pozn.: Řetězec „SUPER_SECRET=abc132“ detekován nebyl, z toho důvodu, že v defaultní konfiguraci Gitleaks hledá klíče o délce 10 až 150 znaků [56].

5.2.5 Fáze nasazení

Před nasazením do produkčního prostředí je nezbytné provést finální bezpečnostní kontroly:

- **Skenování finálních artefaktů:** Zkontrolovat build artefakty na přítomnost zranitelností a malwaru.
- **Ověření integrity:** Zkontrolovat digitální podpisy artefaktů, aby bylo zajištěno, že nebyly po sestavení modifikovány.
- **Bezpečná konfigurace:** Zajistit, že cílové prostředí (servery, cloudové služby, kontejnery) je bezpečně nakonfigurováno (minimalizace oprávnění, vypnutí nepotřebných služeb, blokování portů).
- **Správa citlivých informací:** Zajistit bezpečné předání a uložení konfigurací a citlivých informací (API klíče, hesla k databázím) v produkčním prostředí, nikdy je nekódovat napevno.

5.2.6 Příklad bezpečného Dockerfile pro nasazení

Obrázek níže ukazuje, jak je možné bezpečně nakonfigurovat Docker Image. Pro ukázkou je použita naprosto minimální aplikace, obsahující pouze soubor, *server.js*, soubory pro správce balíčků *packages.json* a *package-lock.json* a Dockerfile. Tento příklad využívá pouze jeden balíček – *express*.

```
1 # 1. Fáze
2 # Použití minimalistického node.js obrazu
3 FROM node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e AS builder
4
5 # Definování pracovního adresáře
6 WORKDIR /app
7
8 # Zkopírování souborů definujících balíčků, následná instalace závislostí bez dev dependencies
9 COPY package.json ./
10 COPY package-lock.json ./
11 RUN npm ci --omit=dev
12
13 # Kontrola, zda některý z balíčků neobsahuje známou zranitelnost
14 RUN npm audit --production --audit-level=moderate
15
16 # Zkopírování zbytku kódu
17 COPY . .
18
19 # 2. Fáze
20 # Použití minimální distroless obrazu pro node.js
21 # neobsahuje shell, správce balíčků ani další běžné utility,
22 # drasticky snižuje potenciální plochu útoku
23 FROM gcr.io/distroless/nodejs:18@sha256:b534f9b5528e69baa7e8caf7bcc1d93ecf59faa15d289221decf5889a2ed3877
24
25 # Definování pracovního adresáře
26 WORKDIR /app
27
28 # Zkopírování nezbytných artefaktů
29 COPY --from=builder /app/node_modules ./node_modules
30 COPY --from=builder /app/server.js ./server.js
31
32 # Otevření portu 3000
33 EXPOSE 3000
34
35 # Přepnutí na neprivilegovaného uživatele
36 USER nonroot
37
38 CMD ["server.js"]
39
40
```

Obrázek 34 Ukázková konfigurace Docker Image

(zdroj: vlastní)

Tento Dockerfile využívá několik způsobů pro zlepšení bezpečnosti:

- **Pinning verzí:** Přesné verze pro obraz Alpine a distroless nodejs jsou vázané na Sha256, takže vždy bude stažen přesně ten samý obraz.
- **Instalace balíčků:** Pro instalaci balíčků je využíván příkaz `npm ci --omit=dev`, díky využití „`npm ci`“, nikoliv „`npm install`“, se zajistí, že pokud v sestavovacím prostředí existuje soubor `node_modules`, bude nejprve smazán, než začne instalace. Další výhodou je kontrola, zda se shodují balíčky v `package.json` a `package-lock.json`, sestavení selže. Také se používá parametr

`--omit=dev`, který zajistí, že nebudou instalovány balíčky, které jsou určeny pouze pro vývoj.

- **Kontrola balíčků:** Pomocí příkazu *npm audit* se provede kontrola, zda použité balíčky neobsahují nějakou známou zranitelnost. Pro ukázkou jsem použil balíček *express* na verzi 4.17.1, která obsahuje známé zranitelnosti. Tady se chování příkazu může lišit, v závislosti na specifikované úrovni zranitelnosti. Pokud je nalezen zranitelný balíček se stejnou, nebo vyšší úrovní zranitelností, sestavení selže. Pokud jsou však zranitelnosti méně vážné, sestavení proběhne.
- **Využití více fází při sestavení:** Dockerfile je v tomto případě rozdělen do dvou částí. První část má na starost stažení balíčků a následnou kontrolu zranitelných závislostí. Druhá část pak kopíruje pouze připravené balíčky a kód který bude spouštět.
- **Použití distroless obrazu:** Tento obraz je naprosto minimalistický, neobsahuje shell, správce balíčků ani další běžné utility. Tímto je významně sníženo riziko potenciálních útoků.
- **Spuštění s nižším oprávněním:** Dockerfile také specifikuje uživatele *nonroot*, který má nižší oprávnění než *root*, díky čemuž se opět snižuje potenciální plocha útoku.

Při použití `--audit-level=critical`, sestavení proběhne úspěšně:

```
docker build --tag "simple-server" .

[+] Building 3.7s (17/17) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 1.25kB
=> [internal] load metadata for gcr.io/distroless/nodejs:18@sha256:
=> [internal] load metadata for docker.io/library/node:18-alpine@sha256:
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [builder 1/7] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c
=> => resolve docker.io/library/node:18-alpine@sha256:8d6421d663b4c
=> [internal] load build context
=> => transferring context: 280.35kB
=> [stage-1 1/4] FROM gcr.io/distroless/nodejs:18@sha256:b534f9b5528e69baa
=> => resolve gcr.io/distroless/nodejs:18@sha256:b534f9b5528e69baa
=> CACHED [builder 2/7] WORKDIR /app
=> CACHED [builder 3/7] COPY package.json ./
=> CACHED [builder 4/7] COPY package-lock.json ./
=> CACHED [builder 5/7] RUN npm ci --omit=dev --ignore-scripts
=> [builder 6/7] RUN npm audit --production --audit-level=critical
=> [builder 7/7] COPY . .
=> CACHED [stage-1 2/4] WORKDIR /app
=> [stage-1 3/4] COPY --from=builder /app/node_modules ./node_modules
=> [stage-1 4/4] COPY --from=builder /app/server.js ./server.js
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:e7e370349e2bdb66659fc4ce0fc84ea6f99
=> => exporting config sha256:0de3e10eb3e84bac968d0acc0ac2dcdba21f3
=> => exporting attestation manifest sha256:24669a559a30a5c79da16b0
=> => exporting manifest list sha256:e90ea6f6b02c1f6009335361202751
=> => naming to docker.io/library/simple-server:latest
=> => unpacking to docker.io/library/simple-server:latest
```

Obrázek 35 Sestavení obrazu pro Docker při detekci „critical“ zranitelností

(zdroj: vlastní)

Při použití `--audit-level=moderate`, sestavení selže, jelikož se mezi balíčky vyskytuje zranitelný balíček s vyšší úrovní zranitelnosti:

```
docker build --tag "simple-server" .

[+] Building 3.7s (12/16)
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 1.25kB
=> [internal] load metadata for gcr.io/distroless/nodejs:18@sha256:b534f9
=> [internal] load metadata for docker.io/library/node:18-alpine@sha256:8
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [builder 1/7] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b
=> => resolve docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3e
=> [internal] load build context
=> => transferring context: 280.35kB
=> [stage-1 1/4] FROM gcr.io/distroless/nodejs:18@sha256:b534f9b5528e69ba
=> => resolve gcr.io/distroless/nodejs:18@sha256:b534f9b5528e69baa7e8caf7
=> CACHED [builder 2/7] WORKDIR /app
=> CACHED [builder 3/7] COPY package.json ./
=> CACHED [builder 4/7] COPY package-lock.json ./
=> [builder 5/7] RUN npm ci --omit=dev --ignore-scripts
=> ERROR [builder 6/7] RUN npm audit --production --audit-level=moderate
-----
```

Obrázek 36 Sestavení obrazu pro Docker při detekci „moderate“ zranitelnosti

(zdroj: vlastní)

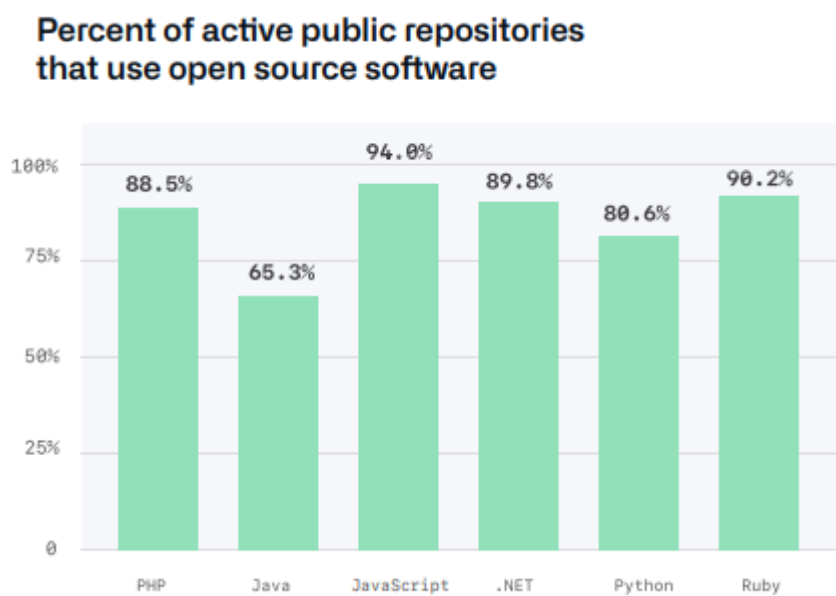
5.2.7 Fáze údržby

Jelikož je bezpečnost kontinuální proces, který nekončí nasazením, je důležité:

- **Monitorování a logování:** Sledovat bezpečnostní události, logy aplikací a infrastruktury pro detekci anomálií a potenciálních útoků
- **Pravidelné aktualizace:** Aplikovat bezpečnostní záplaty na operační systémy, závislosti a samotnou aplikaci. Včasná aktualizace známých zranitelností je kritická.
- **Opakované testování:** Pravidelně provádět bezpečnostní skeny (SAST, DAST) a penetrační testy pro odhalení nových zranitelností
- **Plán reakce na incidenty:** Mít připravený a otestovaný plán pro případ bezpečnostního incidentu, včetně kroků specifických pro kompromitaci dodavatelského řetězce.

5.3 Proaktivní správa a zabezpečení závislostí

Moderní softwarové aplikace jsou z velké části tvořeny externími komponentami – open source knihovnami, frameworky a dalšími balíčky. Studie ukazují, že open-source kód používá přibližně 70-90 % veřejných repositářů.



Obrázek 37 Aktivní veřejné repositáře využívající open-source software

(zdroj: Microsoft [1])

Tato rozsáhlá závislost na externích zdrojích představuje významné riziko v rámci softwarového dodavatelského řetězce. Zranitelnosti v jediné populární knihovně, nebo škodlivý kód záměrně vložený do balíčku mohou mít katastrofální dopad na tisíce aplikací.

Proaktivní správa a zabezpečení těchto závislostí je proto naprosto zásadní součástí strategie zmírnění rizik

5.3.1 Význam inventarizace a sledování závislostí

Základním kamenem správy závislostí je vědět, jaké komponenty aplikace vlastně obsahuje. Software Bill of Materials (SBOM) je formální záznam, který poskytuje detailní inventář softwarových komponent, jejich verzi, licenci a vztahů v rámci softwarového produktu.

SBOM přináší transparentnost do „černé skříňky“ softwarových závislostí a slouží jako základ pro:

- **Správu zranitelností:** Umožňuje rychle identifikovat, zda je aplikace ovlivněna nově objevenou zranitelností v některé z jejích komponent
- **Správu licencí:** Pomáhá zajistit soulad s licenčními podmínkami open-source komponent
- **Analýzu rizik dodavatelského řetězce:** Poskytuje přehled o původu komponent a umožňuje hodnotit rizika spojená s jednotlivými dodavateli.
- **Zajištění integrity:** Může být použit k ověření, že nasazený software obsahuje přesně ty komponenty, které byly schváleny a prověřeny.

Existuje několik standardizovaných formátů pro SBOM, mezi nejrozšířenější patří SPDX (Software Package Data Exchange) a CycloneDX, které umožňují strojové zpracování a sdílení informací o komponentách. Tvorba a správa SBOM se stává stále častěji nejen doporučenou praxí, ale i regulačním požadavkem (např. v návaznosti na US Executive Order 14028) [57].

5.3.2 Generování a správa SBOM

SBOM lze generovat pomocí specializovaných nástrojů, které analyzují projektové soubory (např. package-lock.json, pom.xml, requirements.txt) nebo přímo binární artefakty či kontejnerové obrazy. Mezi populární nástroje patří:

- **Syft (Anchore):** Generuje SBOM z kontejnerových obrazů a souborových systémů
- **Trivy (Aqua Security):** Kromě skenování zranitelností zvládne také generovat SBOM
- **OWASP CycloneDX:** Nabízí nástroje pro různé ekosystémy

Příklad generování SBOM pomocí nástrojů Trivy a CycloneDX-npm na lokálním node.js projektu:

- **Trivy:**

```
trivy fs -format cyclonedx -output sbom.trivy.cdx.json
```

- **CycloneDX-npm:**

```
npx cyclonedx-npm --output-file sbom.cyclonedx-npm.cdx.json
```

Výsledkem jsou soubory, obsahující strukturovaný seznam všech závislostí – včetně tranzitivních, které můžeme dále analyzovat, zjistit, jak spolu jednotlivé závislosti souvisí, jaké jsou verze nalezených závislostí, a následně například zkontrolovat, zda použité verze pro jednotlivé závislosti neobsahují nějakou známou zranitelnost.

Generování SBOM by mělo být integrováno do CI/CD pipeline, aby byl zajištěn aktuální přehled o komponentách pro každý build. Vygenerované SBOM by měly být ukládány a spravovány, například pomocí platformy jako je OWASP Dependency Track. Tato platforma umožňuje kontinuálně monitorovat komponenty na nově objevené zranitelnosti a také spravovat licenční rizika.

Přidání generace SBOM do CI/CD není nic složitého, jako jednoduchý příklad může posloužit ukázka kódu níže, který vygeneruje SBOM a následně jej nahrává na GitHub jako artefakt. Například zde by bylo druhý krok upravit tak, aby vygenerovaný SBOM nahrál do platformy Dependency Track.

```
- name: Generate SBOM
  uses: aquasecurity/trivy-action@0.20.0
  with:
    scan-type: 'fs'
    scan-ref: '.'
    format: 'cyclonedx'
    output: 'sbom.cyclonedx.json'
    exit-code: '0'
    severity: 'UNKNOWN,LOW,MEDIUM,HIGH,CRITICAL'
    ignore-unfixed: 'false'

- name: Upload SBOM artifact
  uses: actions/upload-artifact@v4
  if: always()
  with:
    name: sbom-cyclonedx
    path: sbom.cyclonedx.json
    retention-days: 1
```

Obrázek 38 Krok pro generování a nahrání SBOM jako artefaktu

(zdroj: vlastní)

5.3.3 Skenování závislostí na známe zranitelnosti

Skenování závislostí na známe zranitelnosti (Software Composition Analysis – SCA) je proces identifikace open-source komponent v kódu a vyhodnocení jejich bezpečnostních rizik a licenčních implikací.

Tyto nástroje typicky identifikují jak přímé, tak tranzitivní závislosti projektu. Porovnávají nalezené komponenty a jejich verze s databázemi známých zranitelností (např. National Vulnerability Database – NVD) GitHub Advisory Database, a OSV – Open Source Vulnerabilities.

Následně reportují nalezené zranitelnosti, často včetně jejich závažnosti například pomocí skóre CVSS, a také informace o dostupných opravách, případně novější opravené verze. SCA je klíčové pro odhalování známých bezpečnostních slabin v použitých externích knihovnách, což je jedno z častých rizik v dodavatelském řetězci.

5.3.4 Implementace SCA nástrojů

Existuje spousta SCA nástrojů jak komerčních, tak open source. Mezi tyto patří například:

- **OWASP Dependency-Check:** Populární open source nástroj, podporuje mnoho programovacích jazyků.
- **Trivy:** Kromě SBOM umí i SCA skenování souborových systémů a kontejnerových obrazů.
- **Snyk Open Source:** Komerční nástroj s rozsáhlou databází zranitelností a integracím (nabízí jak placený, tak i bezplatný plán).
- **Integrované nástroje:** Správci balíčků často obsahují vlastní SCA funkce (např. npm audit, pip audit, yarn audit).
- **GitHub Dependabot:** Automaticky skenuje závislosti v repositářích na GitHubu a může navrhnout Pull Requesty s opravami.

Nejjednodušší způsob, jak tohle můžeme vyzkoušet je použít například:

```
npm audit
```

Tento příkaz zkontroluje závislosti definované v *package-lock.json* oproti výchozí databázi zranitelností a vypíše nalezené problémy, jejich závažnost a případně návrhy na opravu.

```
npm audit
# npm audit report

jquery <=3.4.1
Severity: moderate
Potential XSS vulnerability in jQuery - https://github.com/advisories/GHSA-jpcq-cgw6-v4j6
Potential XSS vulnerability in jQuery - https://github.com/advisories/GHSA-gxr4-xjj5-5px2
fix available via `npm audit fix`
node_modules/jquery

1 moderate severity vulnerability

To address all issues, run:
npm audit fix
```

Obrázek 39 Ukázka použití příkazu npm audit

(zdroj: vlastní)

Podobně jako generování SBOM, i SCA skenování by mělo být součástí CI/CD pipeline. Pipeline by měla být nakonfigurována tak, aby selhala (nebo alespoň vyvolala varování, případně vyžadovala schválení), pokud jsou nalezeny zranitelnosti překračující definovanou úroveň závažnosti. Nalezené zranitelnosti je potřeba prioritizovat na základě jejich závažnosti, dostupnosti exploitu a relevance pro danou aplikaci a následně je co nejrychleji opravit aktualizací na bezpečnou verzi.

5.3.5 Proces prověřování a schvalování nových závislostí

Pouhé skenování existujících závislostí nestačí. Je nutné zavést formální proces pro prověřování a schvalování před přidáním jakékoli nové externí komponenty do projektu. Tento proces by měl zahrnovat kontrolu:

- **Původu a reputace:** Kdo je autorem/správcem balíčku? Je aktivně udržovaný? Má dobrou reputaci v komunitě?
- **Bezpečnostní historie:** Obsahuje balíček známe zranitelnosti? Jak rychle byly v minulosti opravovány?
- **Licence:** Je licence kompatibilní s politikou vlastního produktu?
- **Nezbytnost:** Je tato komponenta skutečně potřeba? Neexistuje lepší varianta?

Výsledky prověření by měly být zdokumentovány a schváleny určenou osobou nebo týmem.

5.3.6 Strategie pro pravidelné aktualizace a patchování závislostí

Udržování závislostí v aktuálním stavu je klíčové pro zmírnění rizika zneužití známých zranitelností. Strategie by měla zahrnovat:

- **Pravidelné skenování:** Aktivně monitorovat nově vydané verze závislostí a oznámení o zranitelnostech
- **Automatizace:** Využívat nástroje jako Dependabot, které automaticky detekují dostupné aktualizace a vytvářejí Pull/Merge requesty pro jejich aplikaci.
- **Testování:** Důkladně testovat aplikaci po každé aktualizaci závislostí, aby se ověřilo, zda nedošlo k narušení funkčnosti. Automatizované testy jsou zde neocenitelné.
- **Semantic Versioning (SemVer):** Rozumět principům sémantického verzování, avšak slepě na něj nespolehat. I minoritní nebo patch verze mohou teoreticky obsahovat změny které mohou narušit funkčnost produktu.

5.3.7 Obrana proti útokům typu Dependency Confusion a Typosquatting

Jak bylo popsáno v kapitole 5.2, útoky typu Dependency Confusion zneužívají situace, kdy organizace používá interní balíčky se jmény, která nejsou rezervována ve veřejných repositářích. Pokud správce balíčků není správně nakonfigurován, může při hledání závislosti upřednostnit škodlivý balíček s vyšší verzí nahraný útočníkem do veřejného registru, místo legitimního interního balíčku. Typosquatting využívá podobného principu, ale spoléhá na překlepy v názvech populárních veřejných balíčků.

5.3.8 Konfigurace správců balíčků a privátních repositářů

Obrana proti těmto útokům spočívá v kombinaci několika technik:

- **Používání privátního repositáře:** Uchovávat všechny interní balíčky v dedikovaném, neveřejném repositáři
- **Správná konfigurace správce balíčků:** Nastavit správce balíčků tak, aby primárně hledal závislosti v privátním repositáři a explicitně definovat, jak se má chovat při hledání ve veřejných registrech.

- **Použití názvosloví (scopes/namespaces):** Důsledně používat specifické prefixy pro interní balíčky (např. @společnost/ v npm). Tím se snižuje riziko kolize s veřejnými balíčky a usnadňuje se konfigurace správců balíčků.
- **Rezervace názvů:** Pokud je to možné, preventivně zaregistrovat názvy používané pro interní balíčky ve veřejných repositářích, i kdyby obsahovaly jen prázdný nebo zástupný balíček. Tím se zabrání útočníkům, aby tyto názvy mohli zneužít.
- **Striktní pinning verzí:** V konfiguračních souborech, závislostí specifikovat přesné verze závislostí, včetně jejich hashe, místo používání rozsahů verzí (pomocí „^“, „~“). Tím se zabrání automatické instalaci potenciálně škodlivé novější verze z veřejného repositáře.

Implementace těchto proaktivních opatření mohou organizace výrazně snížit riziko kompromitace prostřednictvím svých softwarových závislostí.

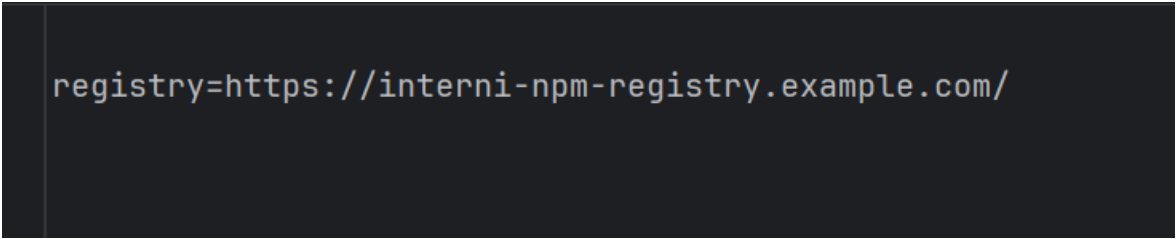
Správná konfigurace souboru .npmrc může vypadat následovně. Definuje, že všechny balíčky pod firemním názvosloví (scope), budou stahovány výhradně z interního registru. Dále definuje registr pro veřejné balíčky. Tato část je dobrovolná, avšak vhodná pro přehlednost.

```
@společnost:registry=https://interni-npm-registry.example.com/  
registry=https://registry.npmjs.org/
```

Obrázek 40 Správná konfigurace .npmrc

(zdroj: vlastní)

Nebezpečná konfigurace .npmrc vypadá následovně. Nastavuje výchozí registr na interní. Při instalaci balíčku se npm podívá na interní registr, ale pokud ho nenajde může začít stahovat balíček se stejným názvem z veřejného registru. Taktéž může použít veřejný registr ve chvíli, kdy je interní registr dočasně nedostupný, nebo vývojář nemá přístup k internímu registru.



```
registry=https://interni-npm-registry.example.com/
```

Obrázek 41 Riziková konfigurace .npmrc

(zdroj: vlastní)

5.4 Zabezpečení infrastruktury a nástrojů dodavatelského řetězce

Zabezpečení samotné infrastruktury a nástrojů, je stejně kritické jako zabezpečení kódu a závislostí. Kompromitace těchto systémů může mít fatální následky.

5.4.1 Zabezpečení systémů pro sestavení a CI/CD pipeline

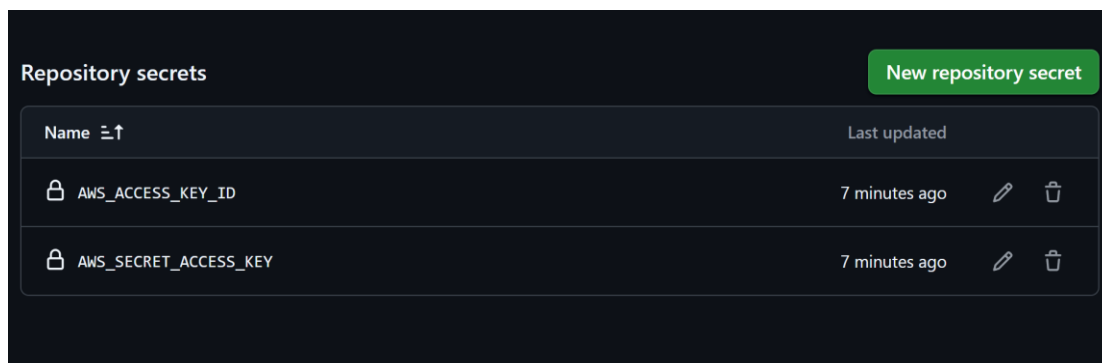
CI/CD pipeline je často centrálním bodem automatizace, a tudíž velmi atraktivním cílem pro útok. Proto je vhodné implementovat princip nejnižších oprávnění. Přístup ke konfiguraci pipeline a k citlivým informacím by měl být striktně omezen. Také je potřeba zamezit jakémukoliv ukládání citlivých údajů přímo v konfiguračních souborech.

```
1  name: example-access-keys.yml
2
3  on:
4    push:
5      branches:
6        - main
7
8  jobs:
9    deploy:
10     runs-on: ubuntu-22.04
11     steps:
12       - name: Checkout code
13         uses: actions/checkout@v4
14
15       - name: Configure AWS Credentials
16         uses: aws-actions/configure-aws-credentials@v4
17         with:
18           aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
19           aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
20           aws-region: eu-north-1
21
22       - name: Deploy to S3
23         run: |
24           echo "Deploying app..."
25
```

Obrázek 42 Ukázka použití secrets v CI/CD

(zdroj: vlastní)

Použití secrets způsobem, jako je na obrázku, je skvělý způsob, jak bezpečným způsobem použít přístupové údaje v GitHub Actions. Avšak aby bylo možné k nim přistoupit, je nutné je také specifikovat v nastavení GitHub.

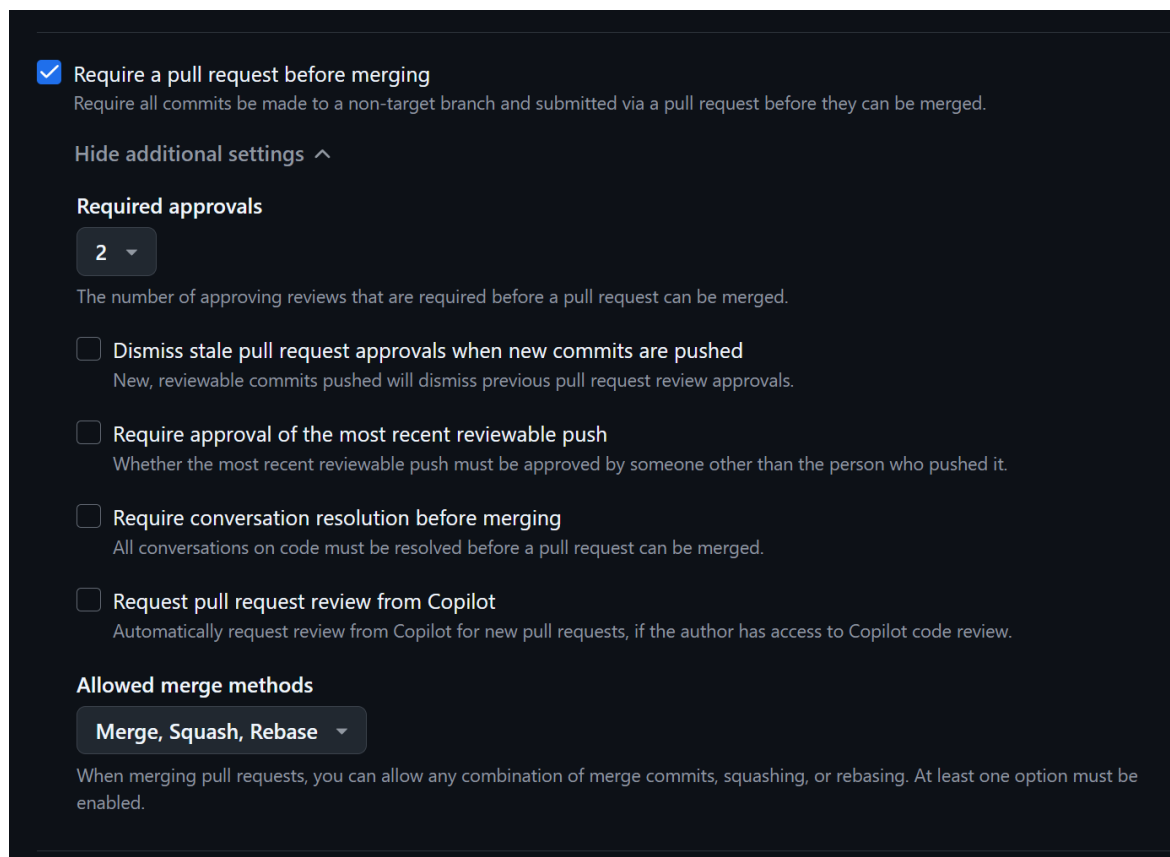


Obrázek 43 Ukázka konfigurace secrets pro GitHub

(zdroj: vlastní)

5.4.2 Zabezpečení repositářů

Pouze zabezpečení CI/CD nestačí, je potřeba pohlídat také samotný repositář, větve, a jasně definovat práva jak pro přístup, tak k provádění různých akcí. Například pokud se vývojář snaží nahrát kód do produkční větve, je vhodné, aby kód prošel přes code review a byl schválen také dalšími vývojáři.



Obrázek 44 Konfigurace GitHubu pro pull requesty

(zdroj: vlastní)

Jako další doporučenou konfigurace, je možné zvážit:

- Zamezení smazání větve
- Vynucení podepsaných commitů
- Vyžadování lineární historie
- Zakázat tzv. Force Push

5.4.3 Zabezpečení vývojářských stanic

Toto téma je extrémně obsáhlé, a bylo by na něj možné napsat samostatnou diplomovou práci. Z toho důvodu jsou níže uvedeny alespoň základní doporučené kroky pro zabezpečení vývojářských stanic.

- Pravidelné aktualizace operačního systému a softwaru
- Používání antivirového řešení
- Zabezpečení přístupu
 - Silná hesla
 - MFA
 - Šifrování disku
 - Uzamykání pracovních stanic při jakémkoliv odchodu od stanice
 - Opatrnost při instalaci nástrojů a rozšíření
- Obezřetnost při nakládání s přihlašovacími údaji

5.5 Verifikace integrity a původu softwaru

I při robustním zabezpečení vývojového procesu a závislostí stále existuje riziko, že software může být útočníkem modifikován během sestavení, uložení, nebo distribuce.

Proto je nezbytné implementovat mechanismy, které umožní ověřit integritu a původ dodávaných artefaktů. Tyto mechanismy pomáhají budovat důvěru v celém dodavatelském řetězci, od vývojáře přes CI/CD až po koncového uživatele.

5.5.1 Digitální podepisování

Digitální podepisování využívá asymetrickou kryptografii, tedy pár veřejného a privátního klíče, k vytvoření elektronického podpisu, který zajišťuje autenticitu, integritu a nepopíratelnost dat. Pro digitální podepsání souboru, artefaktu a podobně, je možné využít například nástroj GNU Privacy Guard. Celý postup od vytvoření klíče, až po jeho ověření je znázorněn na obrázcích níže.

```

zamek@desktop:~/dp-examples$ gpg --full-generate-key
gpg (GnuPG) 2.4.4; Copyright (C) 2024 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
  (1) RSA and RSA
  (2) DSA and Elgamal
  (3) DSA (sign only)
  (4) RSA (sign only)
  (9) ECC (sign and encrypt) *default*
 (10) ECC (sign only)
 (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072)
Requested keysize is 3072 bits
Please specify how long the key should be valid.
    0 = key does not expire
    <n> = key expires in n days
    <n>w = key expires in n weeks
    <n>m = key expires in n months
    <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y

```

Obrázek 45 Vytvoření klíče pomocí GPG

(zdroj: vlastní)

```

zamek@desktop:~/dp-examples$ gpg --output shared-file.sig --detach-sign shared-file
zamek@desktop:~/dp-examples$ ls -la
total 16
drwxr-xr-x  2 zamek zamek 4096 May 25 23:22 .
drwxr-x--- 16 zamek zamek 4096 May 25 23:12 ..
-rw-r--r--  1 zamek zamek   20 May 25 23:13 shared-file
-rw-r--r--  1 zamek zamek  438 May 25 23:22 shared-file.sig
zamek@desktop:~/dp-examples$

```

Obrázek 46 Podepsání souboru pomocí GPG

(zdroj: vlastní)

```

zamek@desktop:~/dp-examples$ gpg --output public-key.asc --armor --export EF7182
zamek@desktop:~/dp-examples$ ls -la
total 20
drwxr-xr-x  2 zamek zamek 4096 May 25 23:24 .
drwxr-x--- 16 zamek zamek 4096 May 25 23:12 ..
-rw-r--r--  1 zamek zamek 2444 May 25 23:24 public-key.asc
-rw-r--r--  1 zamek zamek   20 May 25 23:13 shared-file
-rw-r--r--  1 zamek zamek  438 May 25 23:22 shared-file.sig
zamek@desktop:~/dp-examples$

```

Obrázek 47 Exportování veřejného klíče

(zdroj: vlastní)

```
gpg --import public-key.asc
```

```
gpg: key 9F8492143AF8DA9B: public key "Petr Zamecnik  
gpg: Total number processed: 1  
gpg:             imported: 1
```

Obrázek 48 Importování veřejného klíče

(zdroj: vlastní)

```
gpg --verify shared-file.sig shared-file
```

```
gpg: Signature made 05/25/25 23:22:27 Central Europe Daylight Time  
gpg:       using RSA key EF718274492E92CF8EF40EE09F8492143AF8DA9B  
gpg: checking the trustdb  
gpg: marginals needed: 3 completes needed: 1 trust model: pgp  
gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u  
gpg: Good signature from "Petr Zamecnik <p_zamecnik@utb.cz>" [unknown]  
gpg: WARNING: This key is not certified with a trusted signature!  
gpg:       There is no indication that the signature belongs to the owner.  
Primary key fingerprint: EF71 8274 492E 92CF 8EF4 0EE0 9F84 9214 3AF8 DA9B
```

Obrázek 49 Úspěšné ověření souboru

(zdroj: vlastní)

Z výstupu je vidno, že podpis je technicky správný a soubor opravdu nebyl změněn od doby, kdy byl podepsán, Ale GPG varuje, že si není jistý, zda použitý veřejný klíč opravdu patří tomu správnému člověku.

Pokud by však soubor byl po podepsání upraven, GPG příkaz selže, jak je vidno na obrázku níže

```
gpg --verify shared-file.sig shared-file
```

```
gpg: Signature made 05/25/25 23:22:27 Central Europe Daylight Time  
gpg:       using RSA key EF718274492E92CF8EF40EE09F8492143AF8DA9B  
gpg: BAD signature from "Petr Zamecnik <p_zamecnik@utb.cz>" [unknown]
```

Obrázek 50 Neúspěšné ověření souboru

(zdroj: vlastní)

5.6 Organizační a procesní opatření

Zatímco technická opatření jako skenování závislostí, zabezpečení CI/CD pipeline a podepisování artefaktů tvoří základ ochrany proti útokům na softwarový dodavatelský řetězec, jejich efektivita je velmi závislá na správně nastavených organizačních procesech a bezpečnostním povědomí lidí. Bezpečnost dodavatelského řetězce není pouze technický problém, ale vyžaduje komplexní přístup zahrnující celou organizaci [59].

5.6.1 Definování a vynucování bezpečnostních politik

Je důležité vytvořit, zdokumentovat a komunikovat jasné bezpečnostní politiky specifické pro softwarový vývoj a správu dodavatelského řetězce. Tyto politiky by měly pokrývat následující body:

- **Používání a schvalování externích závislostí:** Definovat kritéria pro výběr, proces prověřování a schvalování nových open source, ale i komerčních komponent. Zde mohou být zahrnuty například schválené (whitelist) nebo zakázané (blacklist) seznamy komponent, nebo licencí.
- **Bezpečné programovací standardy:** Odkazovat na interní nebo externí standardy bezpečného kódu a vyžadovat jejich dodržování, například kontrolou při pull/merge requestech.
- **Konfigurace CI/CD a build systémů:** Stanovit minimální bezpečnostní požadavky pro konfiguraci pipeline, správu hesel či přístupových klíčů.
- **Patch management:** Definovat proces na sledování a aplikaci bezpečnostních aktualizací závislostí.
- **Požadavky na podepisování artefaktů:** Specifikovat, které artefakty musí být podepisovány a jakým způsobem.

Dodržování těchto politik musí být nejen vyžadováno, ale i pravidelně kontrolováno a vynucováno, například pomocí automatizovaných kontrol v CI/CD pipeline nebo prostřednictvím interních auditů.

5.6.2 Pravidelné školení

Lidský faktor je často nejslabším článkem v bezpečnostním řetězci. Pravidelné a cílené školení pro vývojáře, DevOps inženýry, testery a architekty je velmi důležité [60]. Školení by se mělo zaměřit na:

- Aktuální hrozby jako je Dependency Confusion, typosquatting, kompromitace nástrojů atd.
- Principy bezpečného programování a rizika spojená s běžnými chybami.
- Bezpečné používání a správu závislostí.
- Význam a používání bezpečnostních nástrojů (SAST, SDA, DAST).
- Firemní bezpečnostní politiky a postupy.
- Principy sociálního inženýrství, které mohou být použity k získání přístupu k vývojovým systémům.

5.6.3 Procesy pro řízení rizik třetích stran (Vendor Risk Management)

Softwarový dodavatelský řetězec nezahrnuje pouze open source komponenty, ale i komerční software a služby (SaaS, PaaS, IaaS) a případně i externí vývojové týmy. Je důležité zavést robustní proces pro hodnocení a řízení rizik spojených s těmito třetími stranami:

- **Prověřování dodavatelů:** Před navázáním spolupráce je vhodné prověřit bezpečnostní praktiky dodavatele, například pomocí dotazníků, certifikací jako jsou ISO 27001, SOC 2, nebo analýzou jejich reakcí na minulé incidenty.
- **Smluvní záruky:** Zahrnout do smlouvy s dodavatelem specifické bezpečnostní požadavky, práva na audit a povinnosti v případě bezpečnostního incidentu.
- **Průběžné monitorování:** Pravidelně přehodnocovat bezpečnostní stav klíčových dodavatelů.

5.6.4 Vytvoření a testování plánu reakce na incidenty (Incident Response Plan)

I při nejlepší možné prevenci může dojít k bezpečnostnímu incidentu. Organizace musí být připravena rychle a efektivně reagovat. Incident response plán by měl být nastaven tak, aby pokrýval také scénáře útoků na dodavatelský řetězec:

- **Identifikace a analýza:** Jak detekovat kompromitaci závislostí nebo nástroje. Jak rychle analyzovat dopad. Jak využít SBOM pro rychlou identifikace zasažených aplikací.
- **Omezení dopadu:** Jak izolovat postižené systémy? Jak rychle nahradit nebo upravit kompromitovanou komponentu?
- **Odstranění hrozby:** Jak zajistit úplné odstranění hrozby ze systémů, pracovních stanic a případně CI/CD?
- **Obnova:** Jak bezpečně obnovit systémy a data ze záloh? Jak ověřit integritu obnovených systémů?
- **Poučení (Post Incident Activity):** Analyzovat příčiny incidentu a aktualizovat bezpečnostní opatření, aby se předešlo opakování.

Plán reakce na incidenty musí být nejen zdokumentován, ale i pravidelně testován, aby se zajistila jeho aktuálnost, správnost a že odpovědné týmy ví, jak postupovat.

5.7 Praktická ukázka scénáře reakce na incident

Tato část demonstruje aplikace scénáře reakce na incident (Incident Response Plan – IRP) na konkrétní, smyšlený scénář útoku na dodavatelský řetězec softwaru. Scénář představuje reakci na infikování jednoho z nejpopulárnějších Node.js frameworků – Express.js. Cílem je předvést, jak by organizace mohla reagovat na detekci kompromitované závislosti.

Smyslený scénář:

- **Organizace:** Většina webových aplikací a API v organizaci využívá Node.js a framework Express.js.
- **Závislost:** Klíčový balíček `express`, aktuálně používaná verze 4.19.2.
- **Kompromitace:** Neznámý útočník získal přístup k npm účtu jednoho z méně aktivních, ale stále oprávněných vývojářů balíčku `express`. Publikoval novou verzi 4.19.3, která obsahuje nenápadný škodlivý kód. Tento kód se aktivuje pouze za specifických okolností a pokouší se odeslat obsah vybraných requestů, nebo interních proměnných na externí server útočníka.

- **Dopad:** Extrémně vysoký, potenciální únik citlivých dat z mnoha aplikací, kompromitace systémů napříč celou organizací, masivní finanční ztráty a fatální poškození reputace organizace.

5.7.1 Fáze 0 – Před incidentem

- Organizace má nasazené nástroje SCA integrované do CI/CD (Snyk, Trivy).
- Pro každý build se automaticky generuje SBOM a ukládá do centrálního systému pro správu závislostí Dependency Track.
- Existuje definovaný tým pro řešení incidentu (Incident Response Team – IRT) s jasnými rolami a komunikačními kanály.
- Jsou zavedeny bezpečnostní kontroly v CI/CD.

5.7.2 Fáze 1 – Detekce a počáteční analýza

- **Detekce:**
 - **Čas:** středa, 11:30
 - **Událost:** CI/CD pipeline pro jednu z klíčových aplikací spuštěna automaticky po commitu vývojář selhává ve fázi bezpečnostního skenování. Pipeline se pokusila nainstalovat nejnovější dostupnou verzi závislosti *express@4.19.3*. Integrovaný SCA nástroj, který analyzuje použité závislosti okamžitě identifikuje nově publikovanou verzi a generuje kritické upozornění.
 - **Eskalace:** Alert je automaticky eskalován na bezpečnostní tým a DevOps oddělení, aktivuje se celofiremní P1 Incident proces.
- **Počáteční analýza (přibližně 15 minut):**
 - Bezpečnostní tým okamžitě ověřuje upozornění. Rychlá kontrola externích zdrojů potvrzuje nedávné publikování verze 4.19.3 a narůstající podezření komunity na její kompromitaci.
 - Je potvrzeno, že napadený balíček je používán ve více než 80 % backendových služeb organizace.

- Incident je klasifikován jako kritická priorita s potenciálně katastrofálním dopadem, i když kompromitovaná verze zatím nebyla nasazena díky vhodnému nastavení pipeline. Existuje však riziko, že ji mohli stáhnout vývojáři lokálně, nebo do jiné pipeline.
- Je svolán krizový štáb zahrnující tým reakce na incidenty, klíčový DevOps inženýři, vedoucí vývoje a právní oddělení.

5.7.3 Fáze 2 - Omezení šíření

- **Okamžitá akce:**

- DevOps tým okamžitě pozastaví všechny CI/CD pipeline, které by mohly potenciálně stáhnout express.
- DevOps tým také prověří, zda se balíček na infikované verzi nenachází například v interních repositářích jako je AWS CodeArtifact. Je nutné zjistit, zda infikovaná verze nebyla již náhodou stažena a uložena do cache, například jiným procesem před detekcí. Pokud se tak stane, je tato konkrétní verze okamžitě odstraněna například pomocí AWS CLI [61].

```
1  aws codeartifact delete-package-versions \  
2      --domain my-company-domain \  
3      --domain-owner 111122223333 \  
4      --profile target-profile \  
5      --region eu-north-1 \  
6      --repository npm-store \  
7      --format npm \  
8      --package express \  
9      --versions 4.19.3  
10
```

Obrázek 51 Příklad příkazu delete-package-versions

(zdroj: vlastní)

- Dále je vydáno urgentní, celofiremní upozornění pro všechny technické role, zakazující používání express ve verzi 4.19.3 a také nepoužívat příkazy jako *npm install*, nebo *npm update* na projektech kde se nachází express.
- Také je nutné ověřit soubory *package-lock.json* a okamžitě hlásit jakékoliv lokální použití této verze.

- **Identifikace potenciálně zasažených systémů:**
 - Incident response tým analyzuje logy, aby identifikoval všechny požadavky na infikovaný balíček za relevantní období.
 - Analýza CI/CD logů.
 - Využití inventáře SBOM.
 - Komunikace a dotazování vývojářů.

5.7.4 Fáze 3 – Odstranění hrozby

- Analýza infikovaného balíčku: Bezpečnostní tým analyzuje kód infikované závislosti a monitoruje síťový provoz z izolovaných testovacích systémů. Snahou je pochopit, jak mechanismus útoku funguje a identifikovat severity, kam jsou potenciálně odesílána data.
- **Ověření a vyčištění:**
 - Všem relevantním týmům se komunikuje, aby ve všech projektech zajistily striktní pinování verze na 4.19.2 v *package.json*, a hlavně ověřily a případně opravily *package-lock.json*.
 - Probíhá audit lokálních vývojářských prostředí, zda nebyla identifikovaná verze nainstalována. V případě že ano, je nutné tyto stroje považovat za kompromitované.
 - Proběhne kompletní promazání CI/CD cache.
 - Pokud jakákoliv pipeline stihla úspěšně sestavit artefakt s infikovanou verzí, je nutné tyto artefakty bezpečně smazat z registrů.
- **Rotace hesel a klíčů:** Jelikož škodlivý kód mohl odeslat citlivé údaje, jako jsou API klíče, hesla k databázím a tak podobně, je nutná okamžitá a plošná rotace všech potenciálně zasažených přístupových údajů.

5.7.5 Fáze 4 – Obnova

Jakmile je potvrzeno, že všechny projekty používají bezpečnou verzi, včetně explicitního vyžadování verze, která nebyla infikována a *package-lock.json* soubory jsou správné, je

možné postupně obnovovat pozastavené CI/CD pipeline. Přičemž první buildu jsou pečlivě a detailně monitorovány. Nadále také pokračuje monitorování síťového provozu a logů.

5.7.6 Fáze 5 – Poučení z incidentu

Na začátku probíhá analýza, která zkoumá, proč k incidentu došlo a jak rychle reagovaly jednotlivé bezpečnostní týmy a použité nástroje.

Dále se zkoumá, jak je možné zlepšit plán reakce na incident a dalších postupů. V tomto kontextu se může jednat například o zavedení firemních politik, které vynucují schvalovací proces pro aktualizace kritických závislostí.

Mezi další možná zlepšení lze zařadit například:

- **Pinning verzí:** V případě, že by organizace striktně vynucovala pinning verzí, drasticky by kleslo riziko infekce škodlivým balíčkem.
- **Aktualizace IRP:** Upraví se plán reakce na incident, ze získaných poznatků, které byly zjištěny při reakci na tento incident.

5.8 Shrnutí strategie

Navržená strategie pro zmírnění rizik spojených s útoky na softwarový dodavatelský řetězec představuje komplexní, vícevrstvý přístup, který kombinuje jak technická, tak organizační opatření. Cílem není pouze reaktivně řešit incidenty, ale proaktivně budovat odolnost a důvěru v celém životním cyklu softwaru.

Je velmi důležité mít na paměti, že zabezpečení softwarového dodavatelského řetězce není jednorázový úkol, ale kontinuální proces. Vyžaduje neustálou ostražitost, pravidelné hodnocení rizik a adaptaci na neustále se vyvíjející hrozby.

Úspěch této strategie závisí na zapojení a spolupráci napříč celou organizací – od vývojářů a DevOps inženýrů přes bezpečnostní týmy až po management a právní oddělení.

6 SROVNÁNÍ PŘÍSTUPŮ KE SPRÁVĚ ZÁVISLOSTÍ A NÁVRH NEJLEPŠÍCH PRAKTIK

Předchozí kapitoly detailně rozebraly komplexní problematiku útoků na softwarový dodavatelský řetězec a zdůraznily kritickou roli, kterou v těchto útocích hrají softwarové závislosti.

Efektivní správa těchto závislostí se tak stává nejen otázkou efektivity vývoje, ale také klíčovým prvkem bezpečnostní strategie. Softwarový ekosystém však není homogenní, vývojáři a organizace často využívají komponenty ze dvou zásadně odlišných zdrojů. Na jedné straně se jedná o software z komerční sféry a na druhé straně jde o software ze světa open-source. Přístupy ke správě závislostí a rizika se v těchto kontextech významně liší.

Tato kapitola se proto zaměřuje na srovnání a hodnocení praktických přístupů ke správě závislostí pro komerční a open-source software. Cílem je analyzovat klíčové rozdíly v oblastech jako licencování, podpora, transparentnost, bezpečnostní procesy a náklady, ale také na základě tohoto srovnání navrhnout konkrétní, použitelné postupy (best practices) pro efektivní a bezpečnou správu závislostí v obou těchto případech.

6.1 Licencování

Licencování softwarových závislostí představuje kritický, ale často přehlížený aspekt softwarového vývoje. Zatímco zranitelnosti mají přímý bezpečnostní dopad, nedodržení licenčních podmínek může vést k právním a finančním důsledkům.

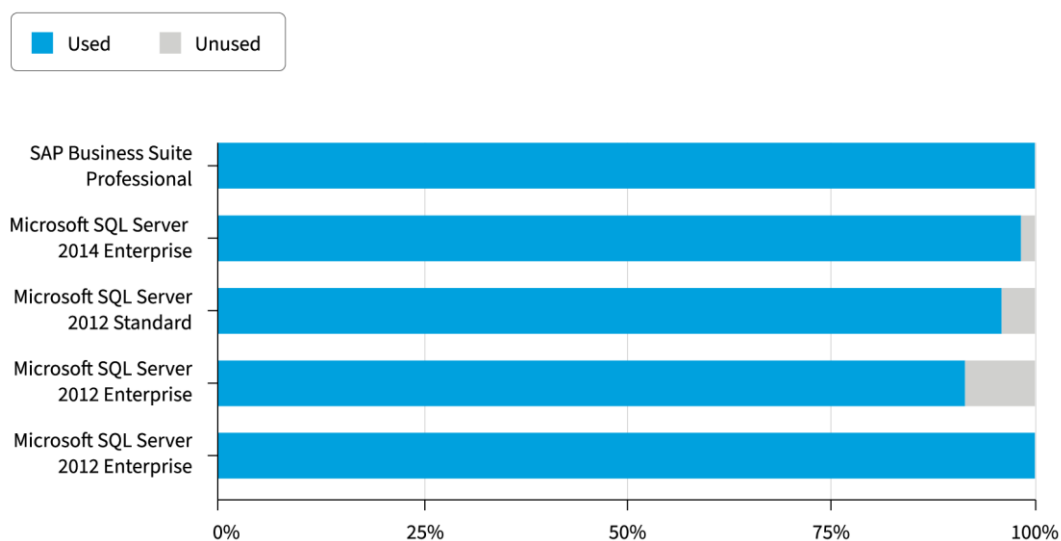
6.1.1 Komerční software a správa licencí

Použití komerčního softwaru je řízeno licenčními smlouvami, nejčastěji ve formě End User License Agreement (EULA). Tyto smlouvy definují podmínky použití jako je počet uživatelů, instalací, omezení na modifikaci či geografické omezení.

Z praktického hlediska je hlavní výzvou zajištění shody – compliance skutečného využití s podmínkami zakoupené licence. Například překročení počtu licencí, nebo nesprávné nasazení může vést k porušení smlouvy.

Ke správě komerčních licencí mohou posloužit nástroje známe jako Software Asset Management (SAM). Mezi ty patří například Flexera One [62], nebo ServiceNow SAM [63].

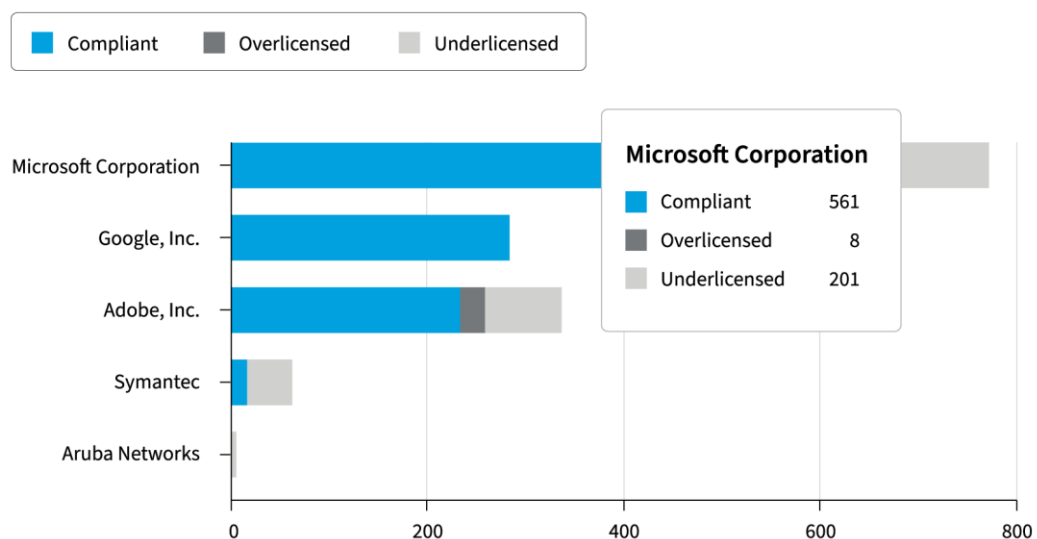
Top 5 applications by cost and their utilization



Obrázek 52 Ukázka Flexera One – využití licencí

(zdroj: <https://www.flexera.com/solutions/software-usage-costs/software-asset-management>)

Application compliance



Obrázek 53 Ukázka Flexera One – compliance

(zdroj: <https://www.flexera.com/solutions/software-usage-costs/software-asset-management>)

6.1.2 Open-source software

Open-source software se řídí širokou škálou veřejných licencí, kde každá definuje odlišná práva a povinnosti. Klíčový rozdíl oproti komerčním licencím je v tom, že se týká samotného kódu.

- **Permisivní licence (MIT, Apache 2.0 ...):** Tyto licence kladou minimální omezení. Typicky vyžadují pouze zachování původního copyrightu a textu licence. Důležitou stránkou je umožnění integrace do proprietárního softwaru bez nutnosti zveřejnění vlastního zdrojového kódu [64].
- **Copyleft licence (GPL, LGPL, AGPL):** Tyto licence vyžadují, aby jakékoliv **odvozené dílo**, které vznikne modifikací nebo propojením s kódem pod copyleft licencí, bylo licencováno pod stejnou, nebo kompatibilní licencí. Například GPL může vyžadovat zveřejnění **celého** zdrojového kódu aplikace [64].

snyk	Copyleft					Permissive			
	GPL	LGPL	AGPL	CC-BY	NPL	Apache	MIT	BSD	Unlicense
Permissions in addition to commercial use, distribution, modification:									
Patent use	●	●	●	●	●	●	●	●	●
Patent use	●	●	●	●	●	●	●	●	●
Conditions									
Disclose source	●	●	●	●	●	●	●	●	●
License & copyright notice	●	●	●	●	●	●	●	Source	●
Network use is distribution	●	●	●	●	●	●	●	●	●
Same license	●	●	Library	●	File	●	●	●	●
State changes	●	●	●	Some	●	●	●	●	●
Limitations/Disclaimers									
Liability	●	●	●	●	●	●	●	●	●
Warranty	●	●	●	●	●	●	●	●	●
Trademark use	No explicit limitation				●	●	●	●	●

Obrázek 54 Snyk open-source licence

(zdroj: <https://snyk.io/articles/open-source-licenses/>)

6.2 Podpora a údržba

Po výběru a integraci softwarové závislosti práce nekončí. Dlouhodobým aspektem je zajištění její podpory a údržby. To zahrnuje řešení nalezených chyb, aplikaci bezpečnostních záplat a někdy přímou podporou organizace či komunity, v závislosti na původu softwaru.

6.2.1 Podpora a údržba komerčního softwaru

U komerčního softwaru je model podpory založen na smluvním vztahu. Organizace platí za licenci a často i za průběžnou podporu a údržbu. Tato podpora je definována v **Service Level Agreement (SLA)**, který specifikuje dobu odezvy, dostupnost podpory a případně i proces eskalace. V případě nalezení chyby, zjištění zranitelnosti či jiného problém, organizace může postupovat následovně:

1. **Kontakt dodavatele:** Využití oficiálních kanálů pro kontaktování podpory, může jít o ticket systém, e-mail, chat a podobně.
2. **Nahlášení problému:** Je nutné problém popsat detailně, ideálně včetně kroků k reprodukci, případně pokud se jedná o zranitelnost, je vhodné uvést například CVE ID.
3. **Čekání na vyřešení:** Dodavatel je smluvně vázán poskytnout pomoc a problém napravit v definovaném čase, často v závislosti na závažnosti problému. Tento fakt je definován v SLA.

6.2.2 Podpora a údržba open-source softwaru

Model podpory a údržby open-source softwaru je značně odlišný a vyžaduje od organizace proaktivnější přístup a často také interní expertízu. Podpora primárně přichází z komunity vývojářů a uživatelů daného projektu. Při práci s open-source závislostmi by vývojáři měli provádět následující kroky pro údržbu závislostí:

1. **Pravidelná kontrola dostupných aktualizací:** Správci balíčků poskytují příkazy pro zjištění, které závislosti mají dostupné novější verze. Níže je ukázka použití příkazu *npm outdated*, avšak organizace by se měli také zaměřit na automatizaci tohoto kroku.

```
~\Documents\UTB\diplomova-prace\prakticka-cast\simple-server git:(main) (0.686s)
npm outdated
Package Current Wanted Latest Location Depended by
express 5.0.0 5.0.0 5.1.0 node_modules/express simple-server
```

Obrázek 55 Použití npm outdated

(zdroj: vlastní)

2. **Řešení problémů a hledání podpory:** Při nalezení chyby, nebo hledání pomoci se vývojáři mohou obrátit na komunitní zdroje:
 - 1) Dokumentace projektu
 - 2) Issue trackery – například GitHub Issues
 - 3) Fóra a Q&A stránky
3. **Kontrola zranitelností:** Bezpečnostní údržba open-source závislostí spočívá v detekci známých zranitelností a aplikaci dostupných oprav. Tento krok je také vhodné automatizovat, alespoň část pro detekování rizika. Níže je ukázka použití *npm audit*.

```
~\Documents\UTB\diplomova-prace\prakticka-cast\simple-server git:(main) (1.077s)
npm audit
# npm audit report

cookie <0.7.0
cookie accepts cookie name, path, and domain with out of bounds characters - https://github.com/advisories/GHSA-pxg6-pf52-xh8x
fix available via `npm audit fix --force`
Will install express@5.1.0, which is outside the stated dependency range
node_modules/cookie
  express 3.0.0-alpha1 - 4.21.0 || 5.0.0-alpha.1 - 5.0.0
  Depends on vulnerable versions of cookie
  node_modules/express

2 low severity vulnerabilities

To address all issues, run:
npm audit fix --force
```

Obrázek 56 Použití npm audit

(zdroj: vlastní)

6.3 Transparentnost kódu

Transparentnost zdrojového kódu představuje jeden z největších rozdílů mezi komerčním a open-source softwarem a má významné dopady na správu závislostí, bezpečnost, údržbu a flexibilitu. V kontextu softwarového dodavatelského řetězce se schopnost nahlédnout do kódu přímo projevuje v možnostech auditu, detekce problémů a budování důvěry.

6.3.1 Komerční software

Zdrojový kód komerčního softwaru je ve většině případech proprietární a pro koncového uživatele či organizaci nedostupný. Hlavním důsledkem je nemožnost provést nezávislý, interní audit kódu na přítomnost chyb, zranitelností nebo zadních vrátek. Organizace je v podstatě nucena plně důvěřovat dodavateli a jeho interním procesům pro zajištění kvality a bezpečnosti. Protože není možné zkontrolovat kód, organizace spoléhají na jiné mechanismy pro posouzení rizika komerčních závislostí:

1. **Audit dodavatele (Vendor Risk Management):** Prověřování bezpečnostních praktik dodavatele, jeho reputace, historie incidentů a procesů.
2. **Smluvní záruky:** Zahrnutí požadavků na bezpečnost a včasné opravy zranitelností přímo do smlouvy.
3. **Certifikace a shoda s normami:** Dodavatel může doložit shodu s mezinárodními normami (např. ISO 27001) nebo získané bezpečnostní certifikace (např. SOC 2 report), které za pomoci externích auditů potvrzují úroveň zabezpečení dodavatele.
4. **Analýza binárního kódu:** V některých specifických případech, pokud je to technicky možné a legální dle EULA, lze provést analýzu kompilovaného kódu nebo sledovat chování aplikace za běhu (např. pomocí dynamické analýzy – DAST, viz. 5.2.4), avšak tyto metody jsou obecně složitější a méně kompletní než analýza zdrojového kódu.

6.3.2 Open-source software

Hlavním znakem open-source softwaru je veřejná dostupnost zdrojového kódu. Kdokoli si kód může stáhnout, prohlédnout, analyzovat a dokonce modifikovat – samozřejmě na licenci. Tato transparentnost umožňuje řadu činností a testování, které nejsou u komerčního software možné, alespoň ve většině případech.

1. **Nezávislý bezpečnostní audit a komunitní kontrola:** Bezpečnostní experti a vývojáři mohou kód prozkoumat, hledat chyby a zranitelnosti. Tento proces může vest k rychlejšímu odhalení a opravě bezpečnostních problémů.
2. **Vlastní oprava chyb:** Pokud organizace narazí na kritickou chybu, nebo zranitelnost a čekání na oficiální opravu je příliš dlouhé, může si díky dostupnosti chybu opravit sama. Z praktického hlediska by tento postup mohl vypadat následovně:
 - a. Vytvořit lokální fork repozitáře knihovny.
 - b. Implementovat opravu chyby.
 - c. Vytvořit vlastní verzi knihovny s touto opravou.
 - d. Ve svém projektu změnit závislost tak, aby odkazovala na opravenou verzi např. za použití interních repozitářů.
3. **Využití nástrojů pro statickou analýzu kódu (SAST):** Transparentnost kódu je základním předpokladem pro efektivní SAST analýzu, viz ukázka v 5.4.2.

6.4 Cyklus vydávání nových verzí a opravy

Způsob, jakým jsou vydávány nové verze bezpečnostní opravy se liší a přímo ovlivňuje plánování a stabilitu.

6.4.1 Komerční software

Vývoj a vydávání komerčního softwaru se obvykle řídí předem definovaným plánem, tzv. release roadmap, který stanovuje dodavatel. Tento plán zahrnuje vydávání hlavních verzí s novými funkcemi, vedlejších verzí s menšími vylepšeními a opravných verzí pro řešení bezpečnostních zranitelností.

Značnou nevýhodou tohoto modelu může být závislost na dodavateli. Pokud dodavatel nepovažuje určitou chybu nebo zranitelnost za prioritu, může vydání opravy trvat déle, než je žádané.

6.4.2 Open-source software

Cykly vydávání open-source softwaru jsou mnohem rozmanitější. Některé velké a dobře organizované projekty, jako jsou například distribuce Linuxu, zejména Ubuntu, nebo

samotné jádro Linuxu, mají pravidelné a předvídatelné plány vydání. Spousta jiných projektů však vydává nové verze nepravidelně, v závislosti na dostupnosti nových funkcí, oprav nebo času vývojářů.

Tato flexibilita může být výhodou, protože umožňuje rychlejší reakci na potřeby uživatelů nebo rychlejší vydání bezpečnostních oprav. Na druhou stranu, nepředvídatelnost může komplikovat dlouhodobé plánování a správu závislostí, zejména pokud organizace spoléhá na mnoho open-source komponent.

6.5 Náklady pro správu závislostí

Ačkoliv využívání softwarových závislostí přináší značné výhody v podobě rychlejšího vývoje a přístupu k hotovým funkcionalitám, jejich efektivní správa není bez nákladů. Tyto náklady se mohou projevit v různých formách, od přímých finančních výdajů, až po nepřímé náklady spojené s časem vývojářů.

6.5.1 Komerční software

Náklady na komerční software jsou charakterizovány především přímými výdaji. Patří sem:

- **Licenční poplatky:** Mohou být jednorázové, nebo formou pravidelného předplatného. Výše poplatků se liší v závislosti na produktu, dodavateli a licenčním modelu.
- **Poplatky za podporu a údržbu:** Často se účtují jako roční procento z původní ceny licence a zajišťují přístup k technické podpoře a novým verzím softwaru. U předplatného modelu bývají tyto poplatky zahrnuty v ceně [65].

Celkové náklady na vlastnictví komerčního softwaru jsou tedy tvořeny těmito přímými, relativně předvídatelnými náklady. Ačkoliv mohou být vysoké, obvykle zahrnují i garantovanou podporu a údržbu ze strany dodavatele, což může snížit potřebu rozsáhlých interních zdrojů pro řešení problémů, opravy chyb a správu aktualizací.

6.5.2 Open-source software

Základní výhodou open-source softwaru je absence přímých licenčních poplatků – software je obvykle zdarma k použití, modifikaci a distribuci. Tvzení že je OSS „zdarma“ však může být zavádějící, pokud vezmeme v úvahu celkové náklady na vlastnictví, zahrnují řadu nepřímých nákladů:

- **Náklady na interní expertízu:** Organizace potřebují kvalifikované zaměstnance schopné implementovat open-source software, konfigurovat, spravovat, udržovat a zabezpečit jeho využívání.
- **Náklady na nástroje:** Efektivní správa OSS vyžaduje investice do nástrojů pro SCA, generování a správu SBOM, monitorování a správu aktualizací.
- **Náklady na podporu:** Pokud komunitní podpora nestačí, mohou organizace platit za komerční podporu od třetích stran, nebo přímo od správců OSS projektu.
- **Náklady na dodržování licencí:** Zajištění souladu s OSS licencemi, včetně sledování tranzitivních závislostí a plnění specifických povinností, vyžaduje čas a zdroje.
- **Náklady na integraci a přizpůsobení:** Ačkoliv je OSS flexibilní, jeho integrace do stávajících systémů nebo přizpůsobení specifickým potřebám může vyžadovat značné zdroje.

6.6 Návrh „Best Practices“ pro správu závislostí

Efektivní správa závislostí a prevence útoků na dodavatelský řetězec vyžaduje kombinaci nástrojů, postupů, procesu a správných návyků. Zatímco komerční a open-source software mají svá specifika, viz. část 6.1-6.5, mnoho principů je univerzálních.

Tato kapitola se tak zaměřuje na praktické, implementovatelné kroky a nástroje, které mohou organizace a vývojáři využít k posílení bezpečnosti svých softwarových projektů.

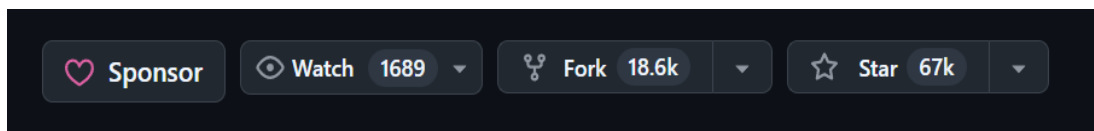
V této kapitole pro praktické ukázky využijí ekosystém Node.js a open-source nástroje, které jsou dostupné zdarma.

6.6.1 Kontrola závislostí před nasazením

Před integrací nové závislosti do projektu je vhodné ji prověřit. Tento krok snižuje riziko zavedení problematické závislosti, ať už z hlediska bezpečnosti nebo udržování.

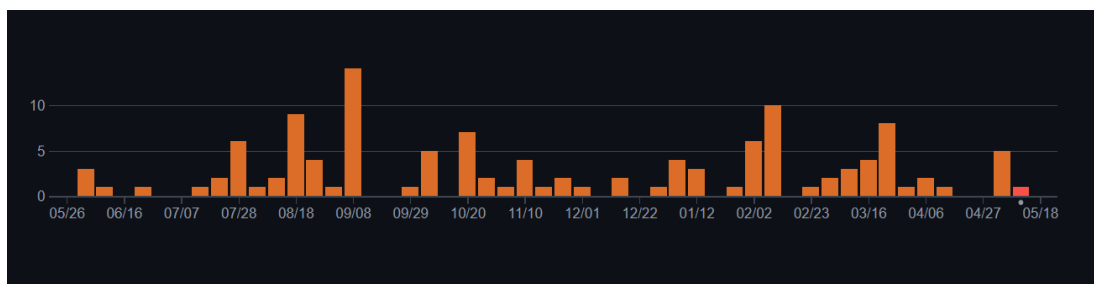
Pokud se jedná o open-source závislost, je dobré zkontrolovat například popularitu, oblíbenost, počet stažení nebo kdo je správcem či přispěvatelem.

Například na GitHubu závislosti express vypadají statistiky následovně:



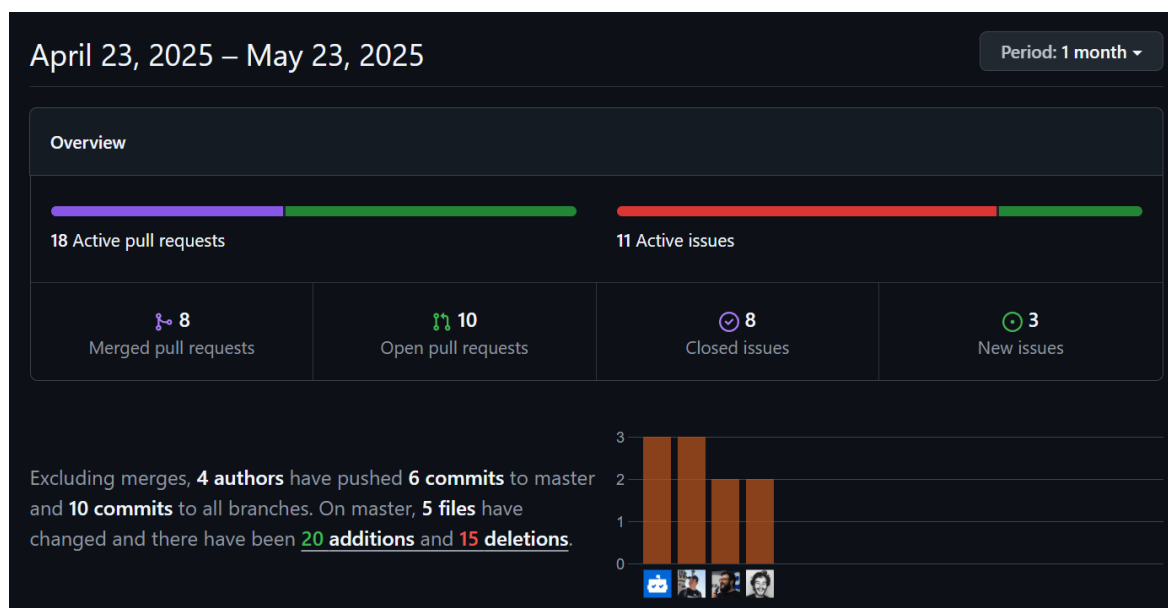
Obrázek 57 GitHub statistika pro express

(zdroj: [66])



Obrázek 58 GitHub statistika pro express – Commits

(zdroj: [67])



Obrázek 59 GitHub statistika pro express – Pulse

(zdroj: [68])

Z těchto statistik vyplývá, že tato závislost je velmi populární, taktéž je stále udržována a má spoustu přispěvatelů. Dá se tedy konstatovat, že použití této závislosti je bezpečné.

Dalším způsobem, jak lze prověřit závislost je využití příkazu „npm view název_závislosti“, který zobrazí informace jako je aktuální verze, licence, počet závislostí, poslední verze nebo také kdo přispívá k vývoji této závislosti.

```
npm view express

express@5.1.0 | MIT | deps: 27 | versions: 283
Fast, unopinionated, minimalist web framework
https://expressjs.com/

keywords: express, framework, sinatra, web, http, rest, restful, router, app, api

dist
.tarball: https://registry.npmjs.org/express/-/express-5.1.0.tgz
.shasum: d31beaf715a0016f0d53f47d3b4d7acf28c75cc9
.integrity: sha512-DT9ck5YIRU+8GYzzU5kT3eHGA5iL+1Zd0Eut0mTE9Dtk+Tvuzd23VBU+ec7HPNS
TxXY055gPV/hq4pSBJDjFpA==
.unpackedSize: 197.0 kB

dependencies:
accepts: ^2.0.0      debug: ^4.4.0      finalhandler: ^2.1.0 on-finished: ^2.4.1
qs: ^6.14.0          serve-static: ^2.2.0
body-parser: ^2.2.0  encodeurl: ^2.0.0  fresh: ^2.0.0      once: ^1.4.0
  range-parser: ^1.2.1 statuses: ^2.0.1
content-type: ^1.0.5 escape-html: ^1.0.3 http-errors: ^2.0.0 parseurl: ^1.3.3
  router: ^2.2.0      type-is: ^2.0.1
cookie: ^0.7.1        etag: ^1.8.1      mime-types: ^3.0.0  proxy-addr: ^2.0.7
  send: ^1.1.0         vary: ^1.1.2
(...and 3 more.)

maintainers:
- wesleytodd <wes@wesleytodd.com>
- jonchurch <npm@jonchurch.com>
- ctcpip <c@labsector.com>
- ulisesgascon <ulisesgascondev@gmail.com>
- sheplu <jean.burellier@gmail.com>

dist-tags:
latest: 5.1.0   latest-4: 4.21.2

published a month ago by wesleytodd <wes@wesleytodd.com>
```

Obrázek 60 Ukázka příkazu npm view

(zdroj: vlastní)

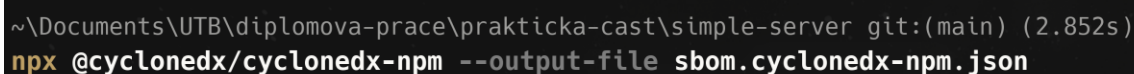
6.6.2 Znalost projektových závislostí a SBOM

Software Bill of Materials (SBOM) je formální inventář softwarových komponent, jejich závislostí a vztahů. Je to základní kámen pro transparentnost a bezpečnost dodavatelského řetězce.

Pro vygenerování SBOM je možné využít hned několik nástrojů, například pro většinu programovacích jazyků existuje nástroje CycloneDX od OWASP Foundation. Fungují velmi dobře, ale mají nevýhodu v tom, že je potřeba použít specifickou verzi softwaru CycloneDX

pro jednotlivé programovací jazyky, například *cyclonedx-npm* nebo *cyclonedx-php-composer*. O něco více se v praxi osvědčil open-source nástroj Trivy vyvíjen společností AquaSecurity, který zvládne generovat SBOM pro škálu různých programovacích jazyků.

Generování SBOM pomocí cyclonedx-npm

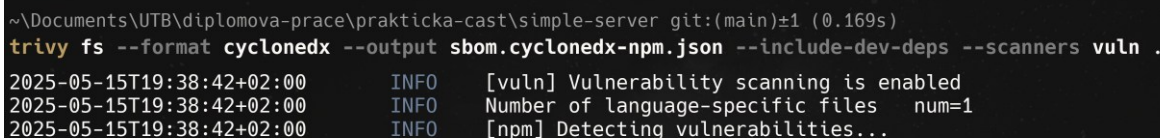


```
~\Documents\UTB\diplomova-prace\prakticka-cast\simple-server git:(main) (2.852s)
npx @cyclonedx/cyclonedx-npm --output-file sbom.cyclonedx-npm.json
```

Obrázek 61 Generování SBOM pomocí cyclonedx-npm

(zdroj: vlastní)

Generování SBOM pomocí Trivy



```
~\Documents\UTB\diplomova-prace\prakticka-cast\simple-server git:(main)±1 (0.169s)
trivy fs --format cyclonedx --output sbom.cyclonedx-npm.json --include-dev-deps --scanners vuln .
2025-05-15T19:38:42+02:00      INFO      [vuln] Vulnerability scanning is enabled
2025-05-15T19:38:42+02:00      INFO      Number of language-specific files  num=1
2025-05-15T19:38:42+02:00      INFO      [npm] Detecting vulnerabilities...
```

Obrázek 62 Generování SBOM pomocí Trivy

(zdroj: vlastní)

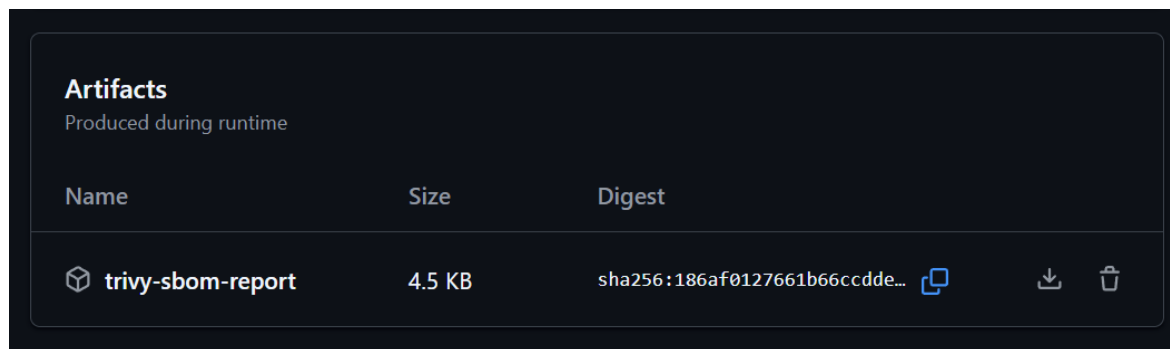
Manuální generování SBOMu je ale vhodné pouze pro ukázkou, v praxi, kde organizace může mít více repositářů či projektů je téměř vždy nutná automatizace. Proto se ukázka níže zaměřuje na využití GitHub Actions pro automatizaci tohoto kroku.

```
1  name: Trivy SBOM generation
2  on:
3    push:
4      branches:
5        - main
6    pull_request:
7      branches:
8        - main
9    workflow_dispatch:
10
11  jobs:
12    generate_sbom:
13      name: Generate SBOM using Trivy
14      runs-on: ubuntu-22.04
15      permissions:
16        contents: read
17      steps:
18        - name: Checkout code
19          uses: actions/checkout@v4
20
21        - name: Generate SBOM using Trivy
22          uses: aquasecurity/trivy-action@master
23          with:
24            scan-type: 'fs'
25            scan-ref: '.'
26            format: 'cyclonedx'
27            output: 'sbom.trivy.cdx.json'
28            exit-code: '0'
29          env:
30            TRIVY_INCLUDE_DEV_DEPS: 'true'
31
32        - name: Upload SBOM artifact
33          uses: actions/upload-artifact@v4
34          with:
35            name: trivy-sbom-report
36            path: sbom.trivy.cdx.json
37            retention-days: 7
38
```

Obrázek 63 Ukázka generování SBOM pomocí Trivy

(zdroj: vlastní)

Tento krok naklonuje kód z repozitáře, poté využije Trivy pro vygenerování SBOM, a následně jej nahraje jako artefakt.



The screenshot shows a dark-themed interface with a table of artifacts. The table has columns for Name, Size, and Digest. One artifact is listed: 'trivy-sbom-report' with a size of '4.5 KB' and a digest starting with 'sha256:186af0127661b66ccdde...'. To the right of the digest are icons for copying, downloading, and deleting.

Artifacts Produced during runtime		
Name	Size	Digest
 trivy-sbom-report	4.5 KB	sha256:186af0127661b66ccdde...   

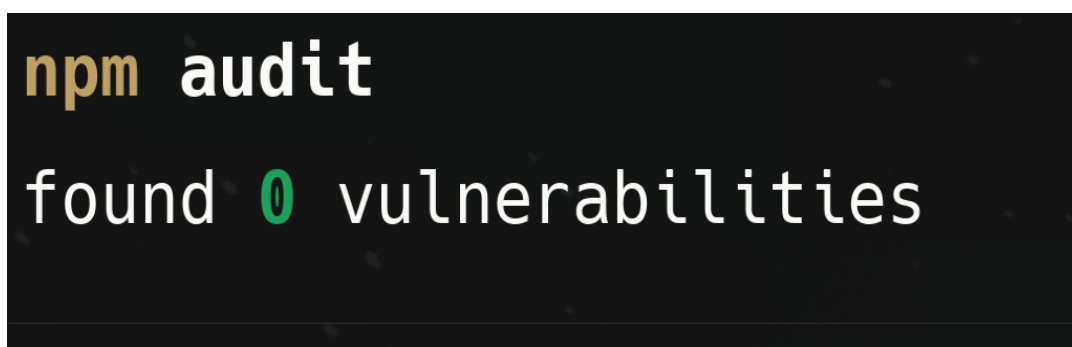
Obrázek 64 Artefakt obsahující SBOM

(zdroj: vlastní)

6.6.3 Aktivní skenování zranitelností

Pravidelné skenování závislostí na známe zranitelnosti – CVE je klíčové, a umožňuje včasnou reakci a nápravu.

- Nástroje správce balíčků



Obrázek 65 Použití npm audit pro skenování zranitelností

(zdroj: vlastní)

- Univerzální nástroje

```
trivy fs --scanners vuln --include-dev-deps .
2025-05-15T20:52:25+02:00      INFO    [vuln] Vulnerability scanning is enabled
2025-05-15T20:52:25+02:00      INFO    Number of language-specific files      num=1
2025-05-15T20:52:25+02:00      INFO    [npm] Detecting vulnerabilities...

Report Summary
```

Target	Type	Vulnerabilities
package-lock.json	npm	0

```
Legend:
- '-': Not scanned
- '0': Clean (no security findings detected)
```

Obrázek 66 Použití Trivy pro skenování zranitelností

(zdroj: vlastní)

Automatické skenování závislostí na zranitelnosti za pomoci GitHub Actions

```
1  name: trivy-vulnerability-scan.yml
2  on:
3    push:
4      branches:
5        - main
6    pull_request:
7      branches:
8        - main
9    workflow_dispatch:
10
11  jobs:
12    vulnerability_scan:
13      name: Scan for vulnerabilities using Trivy
14      runs-on: ubuntu-22.04
15      permissions:
16        contents: read
17        security-events: write
18
19      steps:
20        - name: Checkout code
21          uses: actions/checkout@v4
22
23        - name: Run Trivy
24          uses: aquasecurity/trivy-action@master
25          with:
26            scan-type: 'fs'
27            scan-ref: '.'
28            format: 'table'
29            severity: 'HIGH,CRITICAL'
30            exit-code: '1'
```

Obrázek 67 Trivy sken zranitelností v CI/CD

(zdroj: vlastní)

Výstup po spuštění GitHub Actions

```
138
139 Report Summary
140
141 |-----|
142 | Target | Type | Vulnerabilities | Secrets |
143 |-----|
144 | package-lock.json | npm | 0 | - |
145 |-----|
146 Legend:
147 - '-': Not scanned
148 - '0': Clean (no security findings detected)
149
150 ► Run rm -f trivy_envs.txt
```

Obrázek 68 Výsledek GitHub Actions pro skenování známých zranitelností

(zdroj: vlastní)

6.6.4 Zabezpečení konfigurace správců balíčků

Správná konfigurace může předejít útokům Dependency Confusion, ale také pomáhá zajistit, že jsou závislosti stahovány výhradně z důvěryhodných a schválených zdrojů. Také je minimalizováno riziko nechtěné instalace škodlivého kódu.

1. **Privátní/Scoped registry:** V souboru `.npmrc` v kořenovém adresáři projektu

```
@spolecnost:registry=https://npm.spolecnost.com/
registry=https://registry.npmjs.org/
```

Obrázek 69 Nastavení privátního registru pro interní balíčky

(zdroj: vlastní)

2. **Pinning verzí:** Je vhodné vždy používat lock soubory (package-lock.json). Tyto soubory „uzamknou“ verze všech závislostí, včetně tranzitivních a obsahují také jejich hash.

3. **Instalace balíčků v CI/CD:** Balíčky by se měli v CI/CD instalovat pomocí příkazu *npm ci*, nikoliv *npm install*. Díky tomu budou striktně instalovány verze z lock souboru.

6.6.5 Automatizace bezpečnostních kontrol v CI/CD

Manuální provádění bezpečnostních kontrol je náchylné k chybám, stojí spoustu času a je možné na něj zapomenout. Integrace bezpečnostních skenů a kontrol přímo do CI/CD je zásadní pro konzistentní a včasné odhalování bezpečnostních problémů.

Cíle automatizace v CI/CD jsou následovné:

1. **Včasná detekce:** Odhalit problémy co nejdříve, ideálně již při commitu nebo pull requestu.
2. **Konzistence:** Zajistit, že stejné kontroly jsou aplikovány na veškerý kód.
3. **Prevence:** Blokovat merge, nebo nasazení kódu, který nesplňuje bezpečnostní požadavky nebo standardy.
4. **Zpětná vazba:** Poskytnout vývojářům okamžitou zpětnou vazbu o potenciálních rizicích.

Následující workflow demonstruje integraci několika typů bezpečnostních kontrol:

1. Skenování známých zranitelností v závislostech (SCA)
2. Generování a nahrávání SBOM
3. Statická analýza kódu (SAST)
4. Detekce nechtěně commitnutých citlivých dat

Tato část kódu slouží k pojmenování pipeline, a následně definuje kdy se bude spouštět. Aktuální konfigurace spustí pipeline, pokud detekuje push, nebo pull request na větvi main. Také je možné ji spustit manuálně přímo v GitHubu, díky *workflow_dispatch*. Dále jsou nastaveny práva, jako čtení obsahu, akcí a zápis security eventů.

```
1  name: Security Pipeline
2  on:
3    push:
4      branches:
5        - main
6    pull_request:
7      branches:
8        - main
9    workflow_dispatch:
10
11
12 permissions:
13   contents: read
14   security-events: write
15   actions: read
16
```

Obrázek 70 Bezpečnostní pipeline část 1.

(zdroj: vlastní)

Tato část kódu definuje, na jakém obrazu pipeline poběží, a první dva kroky. Kde první provede checkout kódu, a druhá nastaví prostředí pro Node.js

```
17 jobs:
18   security-checks:
19     name: Run Security Checks
20     runs-on: ubuntu-22.04
21
22   steps:
23     - name: Checkout code
24     ↺ uses: actions/checkout@v4
25       with:
26         fetch-depth: 0
27
28     - name: Setup Node.js
29     ↺ uses: actions/setup-node@v4
30       with:
31         node-version: '24'
32         cache: 'npm'
33
```

Obrázek 71 Bezpečnostní pipeline část 2.

(zdroj: vlastní)

Další krok provede skenování známých zranitelností za pomoci nástroje Trivy. Sken je zaměřen na souborový systém, výstup je ve formátu sarif a kontrola hledá zranitelnosti se závažností **HIGH** a **CRITICAL**. Pokud najde nějakou zranitelnost, nastane chyba, jelikož je nastaven exit-code s parametrem „1“. V tomto kroku je možné ignorovat vývojářské závislosti, jelikož ty se do produkčního kódu nedostanou.

```
34 - name: Scan for vulnerabilities (SCA) with Trivy
35   uses: aquasecurity/trivy-action@master
36   with:
37     scan-type: 'fs'
38     scan-ref: '.'
39     format: 'sarif'
40     output: 'trivy-vulnerability-results.sarif'
41     severity: 'HIGH,CRITICAL'
42     exit-code: '1'
43     ignore-unfixed: 'false'
44   env:
45     TRIVY_SKIP_DEV_DEPS: 'true'
46
```

Obrázek 72 Bezpečnostní pipeline část 3.

(zdroj: vlastní)

Oba kroky níže se věnují SBOMu. V prvním kroku je vygenerován za pomoci nástroje Trivy. Na rozdíl od ukázky výše, zde je naopak žádáno, aby se SBOM generoval včetně vývojářských závislostí, jelikož je vhodné mít přehled o všech závislostech. Druhý krok na obrázku níže slouží pro nahrání SBOMu jako artefaktu, který bude držen po dobu sedmi dnů.

```
46
47   - name: Generate SBOM
48     uses: aquasecurity/trivy-action@master
49     with:
50       scan-type: 'fs'
51       scan-ref: '.'
52       format: 'cyclonedx'
53       output: 'sbom.trivy.cdx.json'
54       exit-code: '0'
55     env:
56       TRIVY_SKIP_DEV_DEPS: 'true'
57
58   - name: Upload SBOM artifact
59     uses: actions/upload-artifact@v4
60     with:
61       name: sbom-report-trivy
62       path: sbom.trivy.cdx.json
63       retention-days: 7
64
```

Obrázek 73 Bezpečnostní pipeline část 4.

(zdroj: vlastní)

Následuje provedení statické analýzy pomocí nástroje ESLint.

```
65 - name: Static Code Analysis (SAST) using ESLint
66   run: |
67     npm install eslint --save-dev
68     npx eslint . --format stylish --max-warnings=0
69
```

Obrázek 74 Bezpečnostní pipeline část 5.

(zdroj: vlastní)

ESLint zde používá následující pravidla, viz obrázek níže. Například využívá doporučená pravidla, povoluje `console.log`, nebo zakazuje funkci `eval()`. V podstatě neexistuje perfektní konfigurace, ale může záležet na více faktorech, jako je bezpečnost vs. komfort, složitost projektu a podobně.

```
rules: {
  ...security.configs.recommended.rules,
  "no-console": "off",
  "no-eval": "error",
  "no-unused-vars": [
    "warn", { "argsIgnorePattern": "^_" }
  ],
  "security/detect-object-injection": "warn",
  "no-secrets/no-secrets": "error",
  "no-unsanitized/method": "error",
  "no-unsanitized/property": "error",
},
```

Obrázek 75 Bezpečnostní pipeline – konfigurace ESLint

(zdroj: vlastní)

Jako finální krok se provede skenování citlivých informací, jako jsou hesla, API klíče a podobně. Zde je využito nástroje Gitleaks.

```
69  
70 - name: Secret scanning using Gitleaks  
71 uses: gitleaks/gitleaks-action@master  
72 env:  
73   GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  
74   GITLEAKS_ENABLE_UPLOAD_ARTIFACT: true  
75   GITLEAKS_ENABLE_SUMMARY: true
```

Obrázek 76 Bezpečnostní pipeline část 6.

(zdroj: vlastní)

Výstup této pipeline pak vypadá následovně. Všechny kroky proběhli úspěšně. Nebyly detekovány žádné citlivé informace jako jsou hesla, nebo API klíče. Také byly vytvořeny dva artefakty, jeden obsahuje report Gitleaks, a druhý obsahuje soubor SBOM.

security-pipeline.yml
on: push

 **Run Security Checks** 35s

Run Security Checks summary

No leaks detected 

[Job summary generated at run-time](#)

Artifacts
Produced during runtime

Name	Size
 gitleaks-results.sarif	6.64 KB
 sbom-report-trivy	6.77 KB

Obrázek 77 Výstup bezpečnostní pipeline

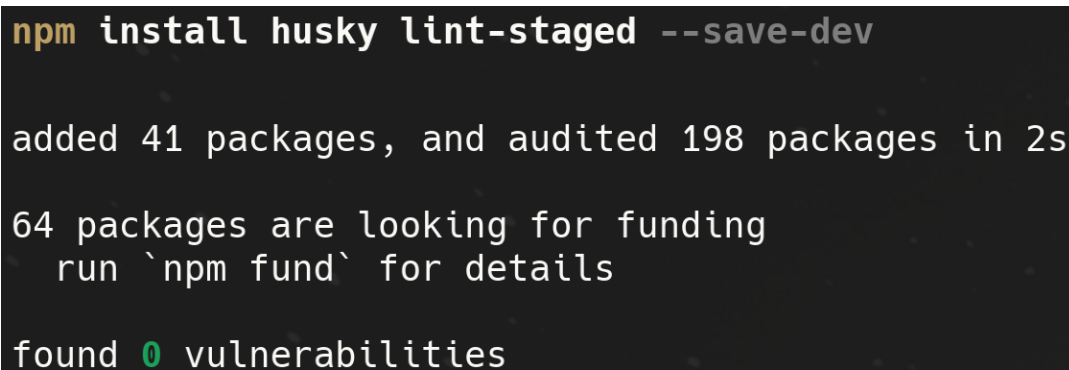
(zdroj: vlastní)

6.6.6 Shift-Left Security

Princip Shift-Left Security spočívá v začleňování bezpečnostních praktik a kontrol co nejdříve do životního cyklu vývoje softwaru. Místo toho, aby se bezpečnost řešila až na konci, např. těsně před nasazením, se klade důraz na odhalení a opravu bezpečnostních problémů již v brzké části vývoje.

Jedním z efektivní způsobů, jak implementovat Shift-Left na úrovni vývojáře, je použití tzv. pre-commit hooků. Pre-commit hooky jsou skripty, které se automaticky spustí před každým commitem a mohou provádět různé kontroly. Pokud některá z kontrol selže, commit je zablokován, do doby, než vývojář problém opraví.

Instalace husky a lint-staged



```
npm install husky lint-staged --save-dev

added 41 packages, and audited 198 packages in 2s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Obrázek 78 Instalace husky a lint-staged

(zdroj: vlastní)

Inicializace husky

```
npx husky init && npm install

> simple-server@1.0.0 prepare
> husky

up to date, audited 198 packages in 1s

64 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Obrázek 79 Inicializace Husky

(zdroj: vlastní)

Konfigurace lint-staged v package.json

```
26   "lint-staged": {
27     "*. {js,jsx,ts,tsx}": [
28       "eslint --fix --max-warnings 0",
29       "npm audit --audit-level=high --prefix ."
30     ],
31     "*": [
32       "gitLeaks protect --staged -v"
33     ]
34   }
```

Obrázek 80 Konfigurace lint-staged

(zdroj: vlastní)

Konfigurace pre-commit souboru

```
3  echo "Running pre-commit checks..."
4  echo "-----"
5
6  echo "Running linter..."
7  npx lint-staged
8  LINT_STAGED_EXIT_CODE=$?
9
10 if [ $LINT_STAGED_EXIT_CODE -ne 0 ]; then
11     echo "Commit rejected -> Issues found."
12     exit 1
13 fi
14
```

Obrázek 81 Pre-commit konfigurace – linter

(zdroj: vlastní)

```
15
16 echo "Scanning for Secrets..."
17 if command -v gitleaks &> /dev/null
18 then
19     gitleaks protect --staged -v --redact
20     GITLEAKS_EXIT_CODE=$?
21
22     if [ GITLEAKS_EXIT_CODE -ne 0 ]; then
23         echo "Commit rejected -> Gitleaks detected secrets."
24         exit 1
25     fi
26
27 else
28     echo "Error: Gitleaks command not found."
29     echo "Please install Gitleaks"
30     exit 1
31 fi
```

Obrázek 82 Pre-commit konfigurace – skenování citlivých informací

(zdroj: vlastní)

```
33
34 echo "Checking dependencies for high vulnerabilities..."
35 npm audit --audit-level=high --prefix .
36 NPM_AUDIT_EXIT_CODE=$?
37
38 if [ NPM_AUDIT_EXIT_CODE -ne 0 ]; then
39     echo "Commit rejected -> High severity vulnerabilities found."
40     exit 1
41 fi
42
43 echo "All pre-commit checks passed."
44 exit 0
```

Obrázek 83 Pre-commit konfigurace – skenování známých zranitelností

(zdroj: vlastní)

Testování funkčnosti pre-commitu:

1. Porušení pravidel ESLintu

```
git commit -m "pre-commit test"
Running pre-commit checks...
-----
Running linter...
✓ Backed up original state in git stash (803a418)
⚠ Running tasks for staged files...
  > package.json – 1 file
  > *.{js,jsx,ts,tsx} – 1 file
    ✗ eslint --fix --max-warnings 0 [FAILED]
    ■ npm audit --audit-level=high --prefix .
  ✓ * – 1 file
↓ Skipped because of errors from tasks.
✓ Reverting to original state because of errors...
✓ Cleaning up temporary files...

✗ eslint --fix --max-warnings 0:

C:\Users\zamec\Documents\UTB\diplomova-prace\praktic
7:1  error  eval can be harmful  no-eval

✗ 1 problem (1 error, 0 warnings)
husky - pre-commit script failed (code 1)
```

Obrázek 84 Pre-commit – porušení pravidel ESLint

(zdroj: vlastní)

2. API klíč nalezen pomocí ESLintu při použití: *"no-secrets/no-secrets": "error"*

```
✖eslint --fix --max-warnings 0:
C:\Users\zamec\Documents\UTB\diplomova-prace\prakticka-cas
7:17 error Found a string with entropy 4.46 : "pv_key_
✖1 problem (1 error, 0 warnings)
husky - pre-commit script failed (code 1)
```

Obrázek 85 Pre-commit – API klíč odhalen pomocí ESLintu

(zdroj: vlastní)

3. API klíč nalezen díky Gitleaks

```
✖gitleaks protect --staged -v:
graph TD
    A(( )) --- B(( ))
    A --- C(( ))
    B --- D[ ]
    C --- D
    D --- E[gitleaks]
    style D fill:none,stroke:none
    style E fill:#ccc,stroke:#333,stroke-width:1px
    subgraph gitleaks
        E
    end

Finding:      const API_KEY = "pv_key_abcdefgh1234567890abcdef1234567890"
Secret:       pv_key_abcdefgh1234567890abcdef1234567890
RuleID:       generic-api-key
Entropy:      4.461092
File:         server.js
Line:         7
Fingerprint:  server.js:generic-api-key:7

11:41PM INF 1 commits scanned.
11:41PM INF scanned ~82 bytes (82 bytes) in 137ms
11:41PM WRN leaks found: 1
```

Obrázek 86 Pre-commit – API klíč odhalen pomocí Gitleaks

(zdroj: vlastní)

4. Selhání kvůli odhalení známé zranitelnosti

```

Checking dependencies for high severity vulnerabilities...
# npm audit report

body-parser <=1.20.2
Severity: high
body-parser vulnerable to denial of service when url encoding
Depends on vulnerable versions of qs
fix available via `npm audit fix --force`
Will install express@4.21.2, which is outside the stated depe
node_modules/body-parser
  express <=4.21.1 || 5.0.0-alpha.1 - 5.0.0
  Depends on vulnerable versions of body-parser
  Depends on vulnerable versions of cookie
  Depends on vulnerable versions of path-to-regexp
  Depends on vulnerable versions of qs
  Depends on vulnerable versions of send
  Depends on vulnerable versions of serve-static
node_modules/express

```

Obrázek 87 Pre-commit – Odhalení známé zranitelnosti

(zdroj: vlastní)

6.6.7 Pravidelné aktualizace a patch management

Používání zastaralých verzí softwarových závislostí je jedním z nejčastějších vektorů útoku. I když byla závislost v době integrace do projektu bezpečná, časem mohou být objeveny nové zranitelnosti. Pravidelná aktualizace závislostí na nenovější bezpečné verze je proto nezbytnou součástí zabezpečení dodavatelského řetězce.

Pro rychlou kontrolu zastaralých závislostí, je možné použít příkaz *npm outdated*.

```

npm outdated

```

Package	Current	Wanted	Latest	Location	Depended by
express	4.17.1	4.17.1	5.1.0	node_modules/express	simple-server

Obrázek 88 Nalezení zastaralé závislosti příkazem „npm outdated“

(zdroj: vlastní)

Manuální sledování všech závislostí a jejich nových verzí je velmi neefektivní a náchylné chybám. K automatizaci může sloužit například nástroj Dependabot.

```
1  version: 2
2  updates:
3    - package-ecosystem: npm
4      directory: /
5      schedule:
6        interval: daily
7      open-pull-requests-limit: 5
8      labels:
9        - "dependabot"
10       - "dependencies"
11
```

Obrázek 89 Konfigurace Dependabot

(zdroj: vlastní)

Dependabot monitoruje databáze zranitelností a nové verze balíčků, ve chvíli, když je to relevantní, vytvoří pull request, který obsahuje následující:

1. Popis změny, např. aktualizace z verze X na verzi Y
2. Informace o kompatibilitě
3. Pokud aktualizace upravuje zranitelnost, odkaz na detail CVE

6.6.8 Závěrečná doporučení a volba nástrojů

Je vhodné začít přímo od základů, tedy mít přehled o všech závislostech, které se v projektu nacházejí. K tomu poslouží SBOM, který obsahuje informace o závislostech včetně těch tranzitivních.

- Verze závislostí, nástrojů a případně také obrazů pro docker či pipeline je vhodné pinovat. Není to vždy nutnost a někde může být vhodné použít nejnovější verzi, avšak je dobré to mít na paměti.
- Také je důležité aktivně hledat zranitelnosti, a nezapomínat na fakt, že pokud závislost není zranitelná dnes, může být zranitelná již zítra.
- Dále je vhodné zabezpečit konfigurace správce balíčků, zejména při použití interních registrů.
- Automatizovat všude, kde je to možné. Jelikož člověk může udělat chybu, nebo na kontrolu úplně zapomenout, je vhodné automatizovat. Automatizace nejen zvýší bezpečnost, ale také šetří čas.
- Velmi důležité je chránit citlivá data, ať už se jedná o API klíče, CI/CD secrets a podobné, nikdy by se neměli psát přímo do zdrojového kódu, ale využívat například „.env“ soubor, CI/CD proměnných nebo správce hesel jako je například 1Password.
- Taktéž je vhodné kontrolovat závislosti ještě před nasazením do vývoje, jak z pohledu bezpečnosti, tak je vhodné ověřit, jak rychle reagují vývojáři či správci na bezpečnostní aktualizace.

Dalším důležitým rozhodnutím je volba nástrojů. Rozhodnutí se často liší v závislosti na povaze projektu, organizace či finančním nebo časovým možnostem. Neexistuje zde perfektní volba a vždy je vhodné zvážit okolnosti.

Tato práce využívá zejména open-source nástroje, které jsou zdarma. Avšak existuje spousta komerčních nástrojů, kterým mohou dát organizace přednost. Může se jednat například o Snyk, Black Duck a podobné.

ZÁVĚR

Tato diplomová práce se zabývala problematikou správy závislostí a prevencí útoků na dodavatelský řetězec v kontextu moderního softwarového vývoje. Cílem práce bylo analyzovat současné hrozby a jejich potenciální dopady, navrhnout strategii a praktická opatření, které mohou organizace a vývojáři implementovat k posílení bezpečnosti softwarových projektů.

V teoretické části byla definována podstata softwarového dodavatelského řetězce. Následně byla provedena analýza rizik a hrozeb, včetně kompromitace vývojových nástrojů, zneužití open-source ekosystému a zranitelností v samotných závislostech. Poté práce popsala hlavní typy útoků, jako tzv. Dependency Confusion, škodlivé závislosti a kompromitaci dodavatelů. Také byly zhodnoceny i potenciální dopady těchto útoků, od finančních ztrát, přes poškození reputace a narušení provozu, až po právní a regulační důsledky.

Praktická část práce se zaměřila na návrh konkrétní strategie pro zmírnění identifikovaných rizik. Tato strategie byla postavena na základních principech jako jsou Zero Trust, Defense in Depth, Shift-Left Security, transparentnost a automatizace. Klíčovou součástí navržené strategie byla proaktivní správa a zabezpečení závislostí, inventarizace, skenování zranitelností a procesy pro prověřování a schvalování nových závislostí. Práce demonstrovala praktické ukázky zabezpečení CI/CD pipeline. V neposlední řadě srovnala přístupy ke správě závislostí pro komerční a open-source software s konkrétními ukázkami.

Závěrem lze konstatovat, že zabezpečení softwarového dodavatelského řetězce je komplexní a náročnou výzvou, která vyžaduje vícevrstvý a proaktivní přístup. Neexistuje jediné univerzální řešení, ale kombinace technických, procesních a organizačních opatření, jak byla navržena a demonstrována v této práci, může významně snížit riziko úspěšného útoku a minimalizovat jeho potenciální dopad. Z toho plyne, že zabezpečení dodavatelského řetězce není jednorázový úkol, ale kontinuální proces vyžadující ostražitost, adaptaci a spolupráci napříč celou organizací.

SEZNAM POUŽITÉ LITERATURY

- [1] JONDOUGLAS. *Osvědčené postupy pro zabezpečení dodavatelský řetězec softwaru* [online]. 1. říjen 2024 [cit. 2025-02-17]. Dostupné z: <https://learn.microsoft.com/cs-cz/nuget/concepts/security-best-practices>
- [2] ESET. *Nejčastější rizika v dodavatelském řetězci a jak se jim vyhnout* [online]. [cit. 2025-02-17]. Dostupné z: <https://digitalsecurityguide.eset.com/cz/nejcastejsi-rizika-v-dodavatelskem-retezci-a-jak-se-jim-vyhnout>
- [3] PAGEWRITER-MSFT. *Doporučení pro zabezpečení životního cyklu vývoje - Microsoft Azure Well-Architected Framework* [online]. 1. září 2024 [cit. 2025-02-17]. Dostupné z: <https://learn.microsoft.com/cs-cz/azure/well-architected/security/secure-development-lifecycle>
- [4] MANUELAP-MSFT. *Doporučení ohledně návrhu dodavatelského řetězce pro vývoj úlohy - Power Platform* [online]. 29. červenec 2024 [cit. 2025-02-17]. Dostupné z: <https://learn.microsoft.com/cs-cz/power-platform/well-architected/operational-excellence/workload-supply-chain>
- [5] ROBERTS, Paul. *Software Supply Chain Attacks: A (partial) History*. *ReversingLabs* [online]. 16. leden 2024 [cit. 2025-02-17]. Dostupné z: <https://www.reversinglabs.com/blog/a-partial-history-of-software-supply-chain-attacks>
- [6] DANSIMP. *Útoky na dodavatelské řetězce - Microsoft Defender for Endpoint* [online]. 29. duben 2024 [cit. 2025-02-17]. Dostupné z: <https://learn.microsoft.com/cs-cz/defender-endpoint/malware/supply-chain-malware>
- [7] *10 Supply chain risks, and ways to mitigate them | SAP Insights* [online]. [cit. 2025-02-17]. Dostupné z: <https://www.sap.com/uk/insights/supply-chain-risks.html>
- [9] *Největší kybernetické útoky a trendy pro 2024: AI, zero trust a IoT*. *MasterDC* [online]. [cit. 2025-02-17]. Dostupné z: <https://www.master.cz/blog/nejvetsi-kyberneticke-utoky-a-trendy-pro-2024-ai-zero-trust-a-iot/>
- [10] ROBOTS, Mobile Industrial. *Zabezpečení dodavatelského řetězce* [online]. 13. říjen 2023 [cit. 2025-02-17]. Dostupné z: <https://mobile-industrial-robots.com/cs/blog/zabezpeceni-dodavatelskeho-retezce>
- [11] HALLER, Martin. *Hacknutí SolarWinds, FireEye a amerického ministerstva financí - Martin Haller, Blog o ochraně a správě firemního IT* [online]. 16. prosinec 2020 [cit. 2025-02-27]. Dostupné z: <https://martinhaller.cz/bezpecnost/hacknuti-solarwinds-fireeye-a-americkeho-ministerstva-financi/>
- [12] *A Timeline of SSC Attacks, Curated by Sonatype* [online]. [cit. 2025-02-17]. Dostupné z: <https://www.sonatype.com/resources/vulnerability-timeline>

- [13] BIRSAN, Alex. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. *Medium* [online]. 29. září 2023 [cit. 2025-02-19]. Dostupné z: <https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610>
- [14] *How Dependency Confusion attack works and How to prevent it | Web Security Lens* [online]. 30. leden 2023 [cit. 2025-02-19]. Dostupné z: <https://www.websecuritylens.org/how-dependency-confusion-attack-works-and-how-to-prevent-it/>
- [15] COHEN, Liad. Preventing Dependency Confusion Attacks. *OX Security* [online]. 9. srpen 2023 [cit. 2025-02-19]. Dostupné z: <https://www.ox.security/understanding-and-preventing-dependency-confusion-attacks/>
- [16] What Is a Dependency Confusion Attack? *Aqua* [online]. [cit. 2025-02-19]. Dostupné z: <https://www.aquasec.com/cloud-native-academy/supply-chain-security/dependency-confusion/>
- [17] *How to Prevent Dependency Confusion* [online]. [cit. 2025-02-19]. Dostupné z: <https://www.activestate.com/blog/how-to-prevent-dependency-confusion/>
- [18] Dependency Confusion Attacks and Prevention: Register Your Private Package Names. *GitGuardian Blog - Take Control of Your Secrets Security* [online]. 15. srpen 2024 [cit. 2025-02-19]. Dostupné z: <https://blog.gitguardian.com/dependency-confusion-attacks/>
- [19] *A Step-by-step Guide to Preventing Dependency Confusion Attacks | Jit* [online]. [cit. 2025-02-19]. Dostupné z: <https://www.jit.io/resources/app-security/preventing-dependency-confusion-attacks>
- [20] *Understanding and Preventing Dependency Confusion - FOSSA* [online]. [cit. 2025-02-19]. Dostupné z: <https://fossa.com/blog/dependency-confusion-understanding-preventing-attacks/>
- [21] *A Pentester's Guide to Dependency Confusion Attacks* [online]. [cit. 2025-02-19]. Dostupné z: <https://www.cobalt.io/blog/a-pentesters-guide-to-dependency-confusion-attacks>
- [22] *Understanding Malicious Dependency in Software Supply Chain | Black Duck Blog* [online]. [cit. 2025-02-24]. Dostupné z: <https://www.blackduck.com/blog/malicious-dependency-supply-chain.html>
- [23] *2024 State of the Software Supply Chain | Executive Summary* [online]. [cit. 2025-02-24]. Dostupné z: <https://www.sonatype.com/state-of-the-software-supply-chain/introduction>
- [24] *SPDX – Linux Foundation Projects Site* [online]. [cit. 2025-02-24]. Dostupné z: <https://spdx.dev/>
- [25] *NVD - Home* [online]. [cit. 2025-02-25]. Dostupné z: <https://nvd.nist.gov/>

- [26] SolarWinds hack explained: Everything you need to know. *WhatIs* [online]. [cit. 2025-02-25]. Dostupné z: <https://www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know>
- [27] Announcements - Data Breach Notification - Forum. *Path of Exile* [online]. [cit. 2025-02-25]. Dostupné z: <https://www.pathofexile.com/forum>
- [28] ANAND, Johnbasco Vijay. *Hardware Cybersecurity: Beyond Software to Secure Devices | NeST Digital* [online]. 1. říjen 2024 [cit. 2025-02-25]. Dostupné z: <https://nestdigital.com/blog/hardware-cybersecurity/>
- [29] Equifax confirms Apache Struts flaw it failed to patch was to blame for data breach. *ZDNET* [online]. [cit. 2025-02-25]. Dostupné z: <https://www.zdnet.com/article/equifax-confirms-apache-struts-flaw-it-failed-to-patch-was-to-blame-for-data-breach/>
- [30] WISENHOFF, Daniel. Vulnerabilities in Dependencies: What You Need to Know. *Debricked* [online]. 2. leden 2021 [cit. 2025-02-25]. Dostupné z: <https://debricked.com/blog/vulnerabilities-dependencies/>
- [31] *Open Source Security and Risk Analysis Report trends | Black Duck* [online]. [cit. 2025-02-25]. Dostupné z: <https://www.blackduck.com/blog/open-source-trends-ossra-report.html>
- [32] What Is a Supply Chain Attack? | CrowdStrike. *CrowdStrike.com* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/supply-chain-attack/>
- [33] *What Is a Supply Chain Attack? | Zscaler* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.zscaler.com/resources/security-terms-glossary/what-is-a-supply-chain-attack>
- [34] *The Continued Increasing Wave of Software Supply Chain Cyber-Attacks* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.netrise.io/xiot-security-blog/the-continued-increasing-wave-of-software-supply-chain-cyber-attacks>
- [35] *What's a Software Supply Chain Attack? Examples and Prevention* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.legitsecurity.com/aspm-knowledge-base/what-is-software-supply-chain-attack>
- [36] Supply Chain Security: Leveraging NDR to Combat Cyberthreats - Progress Flowmon. *Progress Blogs* [online]. 19. březen 2025 [cit. 2025-03-26]. Dostupné z: <https://www.progress.com/blogs/supply-chain-security--leveraging-ndr-to-combat-cyberthreats>
- [37] *How much does a data breach cost? - Embroker* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.embroker.com/blog/cost-of-a-data-breach/>
- [38] Flat Fee Forensic Analysis \$1595, \$1295. *New York Computer Forensics* [online]. [cit. 2025-03-26]. Dostupné z: <https://newyorkcomputerforensics.com/quick-analysis/>

- [39] ZHURER, Yevhen. How to Calculate the Cost of a Data Breach for Your Company — Syteca. *Syteca* [online]. 9. listopad 2021 [cit. 2025-03-26]. Dostupné z: <https://www.syteca.com/en/blog/cost-of-a-data-breach>
- [40] RIBEIRO, Anna. Data breach costs for critical infrastructure sector exceed \$5 million, as time ‘new currency’ in cybersecurity. *Industrial Cyber* [online]. 25. červenec 2023 [cit. 2025-03-26]. Dostupné z: <https://industrialcyber.co/reports/data-breach-costs-for-critical-infrastructure-sector-exceed-5-million-as-time-new-currency-in-cybersecuritydata-breach-costs-for-critical-infrastructure-sector-exceed-5-million-as-time-new/>
- [41] RECORDEDFUTURE. *Ransomware Supply Chain Attacks* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.recordedfuture.com/threat-intelligence-101/cyber-threats/ransomware-supply-chain-attacks>
- [42] CENTRAL, A. S. I. *Ransomware Attack Highlights Threat to Supply Chain Management Providers* [online]. 26. listopad 2024 [cit. 2025-03-26]. Dostupné z: <https://members.asicentral.com/news/industry-news/november-2024/ransomware-attack-highlights-threat-to-supply-chain-management-providers/>
- [43] *Ransomware Costs, a High Price to Pay | Ransomware Containment | Ricoh USA* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.ricoh-usa.com/en/insights/articles/ransomware-costs>
- [44] TOMIMBANG, Sarah Rathke, Kristin Bryan, Gicel. Supply Chains Are The Next Subject of Cyberattacks. *Global Supply Chain Law Blog* [online]. 10. duben 2024 [cit. 2025-03-26]. Dostupné z: <https://www.globalsupplychainlawblog.com/supply-chain/supply-chains-are-the-next-subject-of-cyberattacks/>
- [45] *Supply Chain Cyber Attacks: Understanding Lessons from Major Breaches - Part One* [online]. 26. březen 2025 [cit. 2025-03-26]. Dostupné z: <https://www.aztechit.co.uk/blog/supply-chain-cyber-attacks>
- [46] *How cyberattacks hurt business reputation* [online]. [cit. 2025-03-26]. Dostupné z: <https://www.anapaya.net/blog/5-ways-cyberattacks-can-damage-a-companys-reputation>
- [47] TRAYNOR, Orlaith. The Weakest Link: Safeguarding your Supply Chain Vulnerabilities. *CybelAngel* [online]. 2. leden 2025 [cit. 2025-03-28]. Dostupné z: <https://cybelangel.com/supply-chain-vulnerabilities/>
- [48] MBA, Jason Firsch. The True Cost Of A Data Breach To Small Business Explained. *PurpleSec* [online]. 19. květen 2024 [cit. 2025-03-28]. Dostupné z: <https://purplesec.us/learn/data-breach-cost-for-small-businesses/>
- [49] MOHAN, Vinod. Information Security and The Cost of Non-Compliance | DataCore. *DataCore Software* [online]. 16. prosinec 2024 [cit. 2025-03-28]. Dostupné z: <https://www.datacore.com/blog/information-security-and-cost-of-non-compliance/>

- [50] *Legal Impacts of Data Breaches You Need to Know* [online]. 13. únor 2025 [cit. 2025-03-28]. Dostupné z: <https://www.threatintelligence.com/blog/legal-implications-of-data-breach>
- [51] Software Supply Chain Security: Why Your Organization Must Protect Its Software Supply Chain in 2025. *Secureframe* [online]. [cit. 2025-03-28]. Dostupné z: <https://secureframe.com/blog/software-supply-chain-security>
- [52] KRUTI, Antigoni, Usman BUTT a Rejwan Bin SULAIMAN. *A review of SolarWinds attack on Orion platform using persistent threat agents and techniques for gaining unauthorized access* [online]. B.m.: arXiv. 4. září 2023 [cit. 2025-03-31]. Dostupné z: doi:10.48550/arXiv.2308.10294. arXiv:2308.10294 [cs]
- [53] *gao-22-104746.pdf* [online]. [cit. 2025-03-31]. Dostupné z: <https://www.gao.gov/assets/gao-22-104746.pdf>
- [54] SolarWinds Attack: Play by Play and Lessons Learned. *Aqua* [online]. [cit. 2025-03-31]. Dostupné z: <https://www.aquasec.com/cloud-native-academy/supply-chain-security/solarwinds-attack/>
- [55] *solarwinds_report_2021.pdf* [online]. [cit. 2025-03-31]. Dostupné z: https://www.dfs.ny.gov/system/files/documents/2021/04/solarwinds_report_2021.pdf
- [56] *Semgrep* [online]. [cit. 2025-04-13]. Dostupné z: <https://semgrep.dev/p/gitileaks>
- [57] *How Ortelius Integrates and Consumes SPDX and CycloneDX SBOMs for Enhanced Software Visibility* [online]. 28. říjen 2024 [cit. 2025-04-13]. Dostupné z: <https://ortelius.io/blog/2024/10/28/how-ortelius-integrates-and-consumes-spdx-and-cyclonedx-sboms-for-enhanced-software-visibility/>
- [58] *sigstore/cosign* [online]. Go. B.m.: sigstore. 12. duben 2025 [cit. 2025-04-13]. Dostupné z: <https://github.com/sigstore/cosign>
- [59] *Software Supply Chain Security - OWASP Cheat Sheet Series* [online]. [cit. 2025-04-18]. Dostupné z: https://cheatsheetseries.owasp.org/cheatsheets/Software_Supply_Chain_Security_Cheat_Sheet.html
- [60] FISCHEROVA, Nela. Význam vzdělávání a školení zaměstnanců v oblasti kybernetické bezpečnosti » ARION. *ARION* [online]. 18. listopad 2024 [cit. 2025-04-18]. Dostupné z: <https://arion.cz/aktuality/vyznam-vzdelavani-a-skoleni-zamestnancu-v-oblasti-kyberneticke-bezpecnosti/>
- [61] *delete-package-versions* — *AWS CLI 2.27.2 Command Reference* [online]. [cit. 2025-04-27]. Dostupné z: <https://awscli.amazonaws.com/v2/documentation/api/latest/reference/codeartifact/delete-package-versions.html>

- [62] *Software Asset Management (SAM) Tools: Reduce Software Costs and Risk | Flexera* [online]. [cit. 2025-05-12]. Dostupné z: <https://www.flexera.com/solutions/software-usage-costs/software-asset-management>
- [63] Software Asset Management. *ServiceNow* [online]. [cit. 2025-05-12]. Dostupné z: <https://www.servicenow.com/products/software-asset-management.html>
- [64] Open Source Licenses: Types and Comparison. *Snyk* [online]. [cit. 2025-05-12]. Dostupné z: <https://snyk.io/articles/open-source-licenses/>
- [65] What is the Difference Between Commercial and Open Source? *Intechnic* [online]. [cit. 2025-04-22]. Dostupné z: <https://www.intechnic.com/blog/what-is-the-difference-between-commercial-and-open-source/>
- [66] *expressjs/express: Fast, unopinionated, minimalist web framework for node.* [online]. [cit. 2025-05-24]. Dostupné z: <https://github.com/expressjs/express>
- [67] *Commit Activity · expressjs/express* [online]. [cit. 2025-05-24]. Dostupné z: <https://github.com/expressjs/express/graphs/commit-activity>
- [68] *Pulse · expressjs/express* [online]. [cit. 2025-05-24]. Dostupné z: <https://github.com/expressjs/express/pulse>

SEZNAM OBRÁZKŮ

Obrázek 1 Výstup z příkazové řádky	44
Obrázek 2 Výstup příkazu npm ci.....	45
Obrázek 3 npm audit	45
Obrázek 4 npx license-checker	46
Obrázek 5 Konfigurace Pre-commitu	46
Obrázek 6 Ukázka package.json	47
Obrázek 7 Selhání git commit 1/2.....	47
Obrázek 8 Selhání git commit 2/2.....	48
Obrázek 9 Generování SBOM za pomoci Trivy.....	48
Obrázek 10 Ukázka json souboru zobrazující balíček express	49
Obrázek 11 Ukázka GitHub Actions pro automatizovanou bezpečnostní kontrolu	50
Obrázek 12 Instalace Node.js, závislostí a kontrola zranitelností.....	51
Obrázek 13 Ukázka generování SBOM za pomoci Trivy	51
Obrázek 14 Skenování zranitelností pomocí Trivy.....	52
Obrázek 15 Nahrání SBOM souboru jako artefaktu.....	52
Obrázek 16 Ukázka skenu z GitHub Actions pro kontrolu zranitelností.....	53
Obrázek 17 Ukázka z GitHub Actions při generování SBOM	53
Obrázek 18 Ukázka skenu při kontrole zranitelností	54
Obrázek 19 Nahrání SBOM artefaktu.....	55
Obrázek 20 Zjednodušený model hrozeb pro webovou aplikaci	57
Obrázek 21 Ukázka citlivých informací v kódu	61
Obrázek 22 Ukázka citlivých informací v souboru .env.....	61
Obrázek 23 Ukázka rizikového použití eval	62
Obrázek 24 Ukázka rizikového použití regex	62
Obrázek 25 GitHub Action pro skenování kódu 1/2.....	63
Obrázek 26 GitHub Action pro skenování kódu 2/2.....	63
Obrázek 27 GitHub Action pro skenování zranitelných závislostí.....	64
Obrázek 28 Výsledek ESLint skenu.....	65
Obrázek 29 Výsledek Trivy skenu 1/3.....	65
Obrázek 30 Výsledek Trivy skenu 2/3.....	66
Obrázek 31 Výsledek Trivy skenu 3/3.....	66
Obrázek 32 Výsledek Gitleaks skenu 1/2	67
Obrázek 33 Výsledek Gitleaks skenu 2/2	67

Obrázek 34 Ukázková konfigurace Docker Image	69
Obrázek 35 Sestavení obrazu pro Docker při detekci „critical“ zranitelností	71
Obrázek 36 Sestavení obrazu pro Docker při detekci „moderate“ zranitelností	72
Obrázek 37 Aktivní veřejné repozitáře využívající open-source software	73
Obrázek 38 Krok pro generování a nahrání SBOM jako artefaktu	75
Obrázek 39 Ukázka použití příkazu npm audit	77
Obrázek 40 Správná konfigurace .npmrc	79
Obrázek 41 Riziková konfigurace .npmrc	80
Obrázek 42 Ukázka použití secrets v CI/CD	81
Obrázek 43 Ukázka konfigurace secrets pro GitHub	81
Obrázek 44 Konfigurace GitHubu pro pull requesty	82
Obrázek 45 Vytvoření klíče pomocí GPG	84
Obrázek 46 Podepsání souboru pomocí GPG	84
Obrázek 47 Exportování veřejného klíče	84
Obrázek 48 Importování veřejného klíče	85
Obrázek 49 Úspěšné ověření souboru	85
Obrázek 50 Neúspěšné ověření souboru	85
Obrázek 51 Příklad příkazu delete-package-versions	90
Obrázek 52 Ukázka Flexera One – využití licencí	94
Obrázek 53 Ukázka Flexera One – compliance	94
Obrázek 54 Snyk open-source licence	95
Obrázek 55 Použití npm outdated	97
Obrázek 56 Použití npm audit	97
Obrázek 57 GitHub statistika pro express	102
Obrázek 58 GitHub statistika pro express – Commits	102
Obrázek 59 GitHub statistika pro express – Pulse	102
Obrázek 60 Ukázka příkazu npm view	103
Obrázek 61 Generování SBOM pomocí cyclonedx-npm	104
Obrázek 62 Generování SBOM pomocí Trivy	104
Obrázek 63 Ukázka generování SBOM pomocí Trivy	105
Obrázek 64 Artefakt obsahující SBOM	106
Obrázek 65 Použití npm audit pro skenování zranitelností	106
Obrázek 66 Použití Trivy pro skenování zranitelností	107
Obrázek 67 Trivy sken zranitelností v CI/CD	108
Obrázek 68 Výsledek GitHub Actions pro skenování známých zranitelností	109

Obrázek 69 Nastavení privátního registru pro interní balíčky	109
Obrázek 70 Bezpečnostní pipeline část 1.	111
Obrázek 71 Bezpečnostní pipeline část 2.	112
Obrázek 72 Bezpečnostní pipeline část 3.	113
Obrázek 73 Bezpečnostní pipeline část 4.	114
Obrázek 74 Bezpečnostní pipeline část 5.	115
Obrázek 75 Bezpečnostní pipeline – konfigurace ESLint.....	115
Obrázek 76 Bezpečnostní pipeline část 6.	116
Obrázek 77 Výstup bezpečnostní pipeline.....	117
Obrázek 78 Instalace husky a lint-staged.....	118
Obrázek 79 Inicializace Husky	119
Obrázek 80 Konfigurace lint-staged	119
Obrázek 81 Pre-commit konfigurace – linter.....	120
Obrázek 82 Pre-commit konfigurace – skenování citlivých informací	121
Obrázek 83 Pre-commit konfigurace – skenování známých zranitelností.....	121
Obrázek 84 Pre-commit – porušení pravidel ESLint	122
Obrázek 85 Pre-commit – API klíč odhalen pomocí ESLintu.....	123
Obrázek 86 Pre-commit – API klíč odhalen pomocí Gitleaks.....	123
Obrázek 87 Pre-commit – Odhalení známé zranitelnosti	124
Obrázek 88 Nalezení zastaralé závislosti příkazem „npm outdated“	124
Obrázek 89 Konfigurace Dependabot.....	125

SEZNAM TABULEK

Tabulka 1 Příklady regulačních pokut za narušení dat	36
Tabulka 2 Modelování hrozeb	58

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API	Application Programming Interface (Rozhraní pro programování aplikací)
APT	Advanced Persistent Threat (Pokročilá perzistentní hrozba)
AWS	Amazon Web Services
CI/CD	Continuous Integration/Continuous Delivery or Deployment (Kontinuální integrace/Kontinuální doručování nebo nasazování)
CVE	Common Vulnerabilities and Exposures (Běžné zranitelnosti a expozice)
CVSS	Common Vulnerability Scoring System (Společný systém hodnocení zranitelností)
DAST	Dynamic Application Security Testing (Dynamické testování bezpečnosti aplikací)
DDoS	Distributed Denial of Service (Distribuované odepření služby)
DNS	Domain Name System (Systém doménových jmen)
ECR	Elastic Container Registry (AWS služba pro ukládání Docker obrazů)
EPSS	Exploit Prediction Scoring System (Systém pro predikci skóre zneužití)
ESLint	Nástroj pro statickou analýzu JavaScript/TypeScript kódu
EULA	End User License Agreement (Licenční smlouva s koncovým uživatelem)
GDPR	General Data Protection Regulation (Obecné nařízení o ochraně osobních údajů)
GPG	GNU Privacy Guard
GPL	General Public License (Obecná veřejná licence)
HIPAA	Health Insurance Portability and Accountability Act (Zákon o přenositelnosti a odpovědnosti zdravotního pojištění)
IAM	Identity and Access Management (Správa identit a přístupu)
IaaS	Infrastructure as a Service (Infrastruktura jako služba)
IOT	Internet of Things (Internet věcí)
IP	Internet Protocol (Internetový protokol)
IRP	Incident Response Plan (Plán reakce na incidenty)
IRT	Incident Response Team (Tým reakce na incidenty)
ISO	International Organization for Standardization (Mezinárodní organizace pro normalizaci)
JSON	JavaScript Object Notation
LGPL	Lesser General Public License (Menší obecná veřejná licence)
M&A	Mergers and Acquisitions (Fúze a akvizice)
MFA	Multi-Factor Authentication (Vícefaktorové ověřování)
MSP	Managed Service Provider (Poskytovatel řízených služeb)
NIS2	Network and Information Systems Directive 2 (Směrnice o bezpečnosti sítí a informací 2)
NPM	Node Package Manager (Správce balíčků pro Node.js)
NVD	National Vulnerability Database (Národní databáze zranitelností)
OIDC	OpenID Connect
OSV	Open Source Vulnerabilities (Zranitelnosti v open source)
OSS	Open Source Software (Software s otevřeným zdrojovým kódem)
OWASP	Open Web Application Security Project (Projekt zabezpečení webových aplikací s otevřeným zdrojovým kódem)
PaaS	Platform as a Service (Platforma jako služba)
PyPI	Python Package Index (Index balíčků pro Python)
RCE	Remote Code Execution (Vzdálené spuštění kódu)
ReDoS	Regular Expression Denial of Service (Odepření služby pomocí regulárního výrazu)
SaaS	Software as a Service (Software jako služba)
SAM	Software Asset Management (Správa softwarových aktiv)
SAST	Static Application Security Testing (Statické testování bezpečnosti aplikací)

SBOM	Software Bill of Materials (Seznam softwarových komponent)
SCA	Software Composition Analysis (Analýza složení softwaru)
SDLC	Software Development Lifecycle (Životní cyklus vývoje softwaru)
SLA	Service Level Agreement (Dohoda o úrovni služeb)
SOC 2	System and Organization Controls 2 (Kontroly systémů a organizací 2)
SPDX	Software Package Data Exchange (Formát pro výměnu dat o softwarových balíčcích)
SQL	Structured Query Language (Strukturovaný dotazovací jazyk)
SVR	Sluzhba Vneshney Razvedki (Ruská zahraniční zpravodajská služba)
USD	United States Dollar (Americký dolar)
XSS	Cross-Site Scripting (Skriptování napříč stránkami)

SEZNAM PŘÍLOH

Příloha P I: Ukazkovy-Projekt-1

Příloha P II: Ukazkovy-Projekt-2

Příloha P III: Podepisování