Autorství kódu v éře AI: Srovnávací analýza bezpečnosti softwaru vytvořeného člověkem a umělou inteligencí

Část 1: Shrnutí

Tato zpráva předkládá daty podloženou analýzu, která dochází k závěru, že kód generovaný umělou inteligencí (AI) není ze své podstaty více či méně bezpečný než kód psaný člověkem, ale jeho bezpečnostní profil je *kvalitativně i kvantitativně odlišný*. Ačkoli AI může urychlit vývoj, přináší ve velkém měřítku specifický soubor vysoce závažných zranitelností a vykazuje nebezpečné, protiintuitivní chování, jako je zhoršování bezpečnosti v důsledku iterativního vylepšování.

Klíčové zjištění 1 (Lidský základ): Softwarový ekosystém je již nyní zatížen značným bezpečnostním dluhem. Data ze zprávy společnosti Veracode "State of Software Security" (SoSS) pro rok 2025 ukazují, že 80,3 % aplikací vytvořených člověkem obsahuje alespoň jednu bezpečnostní chybu a téměř 75 % organizací má bezpečnostní dluh.[1] Tím je stanoven chybný, ale známý výchozí stav.

Klíčové zjištění 2 (Profil zranitelností AI): Kód generovaný AI, ačkoliv je strukturálně jednodušší, je náchylný k zavádění většího objemu závažných bezpečnostních zranitelností, zejména v kategoriích jako OS Command Injection (CWE-78) a Hardcoded Credentials (CWE-798), jak dokládá podrobná akademická analýza.[2] Zpráva Veracode GenAI pro rok 2025 to potvrzuje a uvádí, že 45 % kódu generovaného AI obsahuje zranitelnosti, přičemž míra selhání u jazyka Java dosahuje až 70 %.[3]

Klíčové zjištění 3 (Paradox iterace): Bylo identifikováno kritické a neintuitivní riziko: iterativní "vylepšování" kódu pomocí velkých jazykových modelů (LLM) může paradoxně *zvýšit* počet kritických zranitelností. Výzkum ukazuje nárůst kritických chyb o 37,6 % již po pěti iteracích, což zpochybňuje běžný vývojářský postup vylepšování kódu prostřednictvím dialogu s AI.[5]

Strategický imperativ: Přijetí AI ve vývoji softwaru vyžaduje zásadní změnu bezpečnostní strategie. Přístup "důvěřuj, ale prověřuj" je nedostatečný. Organizace musí vyvinout svůj bezpečný životní cyklus vývoje softwaru (Secure Software Development Lifecycle, SSDLC) tak, aby zahrnoval povinné revize kódu vytvořeného AI za účasti člověka (human-in-the-loop), přísné standardy pro tvorbu promptů (prompt engineering) a bezpečnostní nástroje přizpůsobené specifickým vzorcům chyb AI.

Část 2: Stav bezpečnosti v kódu psaném člověkem: Daty podložený výchozí stav

Tato část podrobně kvantifikuje stávající bezpečnostní stav softwarového prostředí a poskytuje základní kontext a srovnávací měřítko, vůči kterému musí být kód generovaný AI posuzován.

2.1 Prevalence a hustota chyb: Endemický stav

Analýza téměř půl milionu aplikací odhaluje, že bezpečnostní chyby nejsou zdaleka vzácné. Zpráva SoSS 2025 od společnosti Veracode zjišťuje, že **80,3 % aplikací má alespoň jednu bezpečnostní chybu**.[1] To dokazuje, že zranitelnost je výchozím stavem pro většinu softwaru. Problémem není jen přítomnost, ale i objem. Typická aplikace má hustotu chyb přibližně

47 chyb na jeden megabajt kódu.[1] Tato metrika normalizuje data s ohledem na velikost aplikace a zdůrazňuje všudypřítomnou povahu nezabezpečeného kódu. Více než polovina (

56,2 %) všech aplikací obsahuje chyby klasifikované jako "vysoce" nebo "kriticky" závažné.[1] To naznačuje, že většina aplikací skrývá rizika, která by mohla vést k významným narušením bezpečnosti.

2.2 Anatomie běžných zranitelností (OWASP & CWE)

Seznam kritických rizik pro webové aplikace OWASP Top 10, který je průmyslovým standardem, se nachází ve **47,7 % aplikací**.[1] Seznam 25 nejnebezpečnějších softwarových slabin (CWE Top 25) je přítomen ve **38,7 % aplikací**.[1]

Dominantní kategorií chyb je **Broken Access Control**. Seznam OWASP Top 10 pro rok 2021 uvádí Broken Access Control jako riziko číslo jedna. Data ukazují, že **94 % testovaných aplikací mělo nějakou formu porušené kontroly přístupu** a CWE (Common Weakness Enumerations) mapované na tuto kategorii měly více výskytů než jakákoli jiná.[9] To poukazuje na systémové selhání v lidmi vedeném návrhu a implementaci autorizační logiky. Navzdory letům zvyšování povědomí zůstávají

Injection chyby hlavní hrozbou, přičemž 94 % aplikací bylo na ně testováno a jejich přidružené CWE jsou druhou nejčastější kategorií.[9]

2.3 Endemická výzva bezpečnostního dluhu

Bezpečnostní dluh – chyby, které zůstávají neopravené déle než rok – představuje kritické obchodní riziko. **74,2 % organizací nahromadilo bezpečnostní dluh**, přičemž polovina (**49,9 %**) má *kritický* dluh (vysoce závažné, dlouhodobě neřešené chyby).[1]

Hlavním přispěvatelem je softwarový dodavatelský řetězec. Zatímco 11 % veškerého bezpečnostního dluhu pochází z kódu třetích stran, toto číslo prudce stoupá na **70 % při pohledu specificky na kritický bezpečnostní dluh**.[1] To ukazuje, že největší neřízená rizika často leží v závislostech, nikoli v kódu první strany. Tento problém je umocněn skutečností, že pouze 49 % organizací má formální bezpečnostní politiku pro používání open-source, což naznačuje významnou mezeru v řízení.[14]

2.4 Překážky a časové rámce nápravy

Medián doby potřebné k opravě 50 % objevených chyb (poločas rozpadu) je více než osm měsíců (**252 dní**).[1] Ohromujících

28 % chyb zůstává otevřených dva roky po objevení.[1] Chyby v knihovnách třetích stran se opravují obtížněji; mají poločas rozpadu 12 měsíců ve srovnání s 8 měsíci u kódu první strany.[1] To je ještě umocněno faktem, že oprava zranitelností v open-source projektech trvá téměř o 20 % déle než v proprietárních projektech.[14]

Výchozí stav pro lidmi psaný kód není stav bezpečnosti, ale stav všudypřítomné, známé a špatně spravované zranitelnosti. Vysoká prevalence chyb, značný bezpečnostní dluh a pomalá náprava vytvářejí ekosystém "zranitelný ve výchozím stavu". Data z Veracode a Snyk konzistentně ukazují vysokou míru chybovosti (přes 80 %) a dlouhé doby oprav (přes 8 měsíců). Data OWASP odhalují, že nejběžnějšími chybami jsou základní logické chyby (Broken Access Control), nikoli exotické exploity. To naznačuje, že i savedenými osvědčenými postupy a nástroji se lidští vývojáři konzistentně potýkají se základy bezpečnosti. Jakékoli srovnání s AI musí tedy uznat, že lidský standard je již nízký. Relevantní otázkou není "je AI dokonalá?", ale "mění AI povahu tohoto již tak nedokonalého systému k lepšímu, nebo k horšímu?".

Softwarový dodavatelský řetězec je epicentrem kritického rizika ve vývoji vedeném člověkem. Skutečnost, že 70 % kritického bezpečnostního dluhu se nachází v kódu třetích stran, je zásadním zjištěním. Vývojáři se silně spoléhají na open-source knihovny pro zvýšení rychlosti, avšak řízení těchto závislostí je slabé (pouze 49 % má zavedenou politiku). Oprava chyb v těchto závislostech je pomalejší a složitější. To vytváří kumulativní efekt, kdy nejrizikovější kód je zároveň nejobtížněji zabezpečitelný. To se stane kriticky důležitým při analýze AI, protože modely AI jsou trénovány na stejném open-source kódu a potenciálně dědí a zesilují jeho rizika.

Metrika	Hodnota	Zdroj(e)
Aplikace s alespoň jednou chybou	80,3 %	[1]
Aplikace s vysoce/kriticky závažnými chybami	56,2 %	[1]
Aplikace s chybami z OWASP Top 10	47,7 %	[1]
Průměrná hustota chyb (na MB)	~47	[1]
Organizace s bezpečnostním dluhem	74,2 %	[1]
Organizace s kritickým bezpečnostním dluhem	49,9 %	[1]
Podíl kritického dluhu z kódu 3. stran	70 %	[1]

Medián doby nápravy 50 % chyb (poločas rozpadu)	252 dní	[1]
Poločas rozpadu pro kód 1. strany vs. 3. stran	8 měsíců vs. 12 měsíců	[1]

Tabulka 1: Klíčové bezpečnostní metriky kódu psaného člověkem (data z let 2024-2025)

Část 3: Vznik generování kódu pomocí AI: Nové paradigma bezpečnostních rizik

Tato část přechází od lidského základu ke specifickým bezpečnostním charakteristikám kódu generovaného AI, přičemž čerpá z cíleného průmyslového výzkumu a raných akademických zjištění.

3.1 Agregovaná míra zranitelností: Systémový problém

Zpráva Veracode GenAI Code Security Report pro rok 2025, která analyzovala více než 100 LLM, zjistila, že **45 % kódu generovaného AI obsahuje bezpečnostní zranitelnosti**.[3] Toto zjištění potvrzují i další studie od Endor Labs (40 %+) a Georgetown's CSET (téměř 50 %).[4]

Riziko není rovnoměrné napříč jazyky. Studie Veracode zjistila, že **70 % úryvků kódu v jazyce Java generovaných AI bylo nezabezpečených**, zatímco Python, C# a JavaScript měly míru selhání mezi 38-45 %.[3] Výzkum naznačuje, že modely se zlepšují v přesnosti kódování, ale nikoli v bezpečnosti, což naznačuje, že se jedná o systémový problém související s trénovacími daty a architekturou modelu, nikoli jen o otázku škálování.[3]

3.2 Dominantní vzorce chyb vyvolaných AI

LLM se potýkají s určitými typy zranitelností. Zpráva Veracode GenAI zdůrazňuje **86% míru selhání u Cross-Site Scripting (CWE-80)** a **88% míru selhání u Log Injection (CWE-117)**.[3] Modely AI jsou trénovány na rozsáhlých datových sadách veřejného kódu, včetně kódu s existujícími chybami. To znamená, že často reprodukují nezabezpečené vzorce, jako je chybějící validace vstupu (CWE-20), SQL injection (CWE-89) a OS command injection (CWE-78).[15] Prompty, které nejsou explicitně zaměřeny na bezpečnost, často vedou ke kódu bez autentizace, s pevně zakódovanými přihlašovacími údaji (CWE-798) nebo s porušenou kontrolou přístupu (CWE-284).[15]

3.3 Nová rizika a nepředvídané důsledky

Významným a novým rizikem je tendence modelů AI navrhovat instalaci softwarových balíčků, které

neexistují. Studie z roku 2024 zjistila, že **přibližně 19,7 % balíčků navržených AI je "halucinovaných"**. Útočníci mohou tyto názvy balíčků zaregistrovat a publikovat škodlivý kód, který nic netušící vývojáři následně nainstalují. Tento útok se nazývá "slopsquatting".[4]

Dalším rizikem je "architektonický posun". AI může zavést jemné změny v návrhu, které porušují bezpečnostní invarianty, aniž by porušovaly syntaxi. Tyto chyby často unikají nástrojům pro statickou analýzu a lidské revizi, protože kód "vypadá správně", ale v širším architektonickém kontextu se chová nezabezpečeně.[15]

Generování kódu pomocí AI nejen automatizuje lidské kódování, ale také industrializuje tvorbu specifických, vysoce závažných zranitelností (XSS, Injection) v masivním měřítku. Zatímco lidský kód má široké rozložení typů chyb, kód generovaný AI vykazuje extrémně vysokou míru selhání u koncentrovaného souboru dobře známých zranitelností, jako jsou XSS a Log Injection (přes 85 %). To naznačuje, že modely AI se systematicky nenaučily vzorce bezpečného kódování pro tyto specifické vektory útoku ze svých trénovacích dat. Zatímco lidský tým může dělat různé chyby, organizace spoléhající na AI zaznamená masivní příliv několika specifických, nebezpečných tříd chyb, což vyžaduje cílenou a odlišnou obrannou strategii.

Trénovací data jsou "prvotním hříchem" bezpečnosti kódu AI. Zranitelnosti přítomné v open-source ekosystému jsou učením, replikací a "praním" prostřednictvím LLM. Jak bylo stanoveno, softwarový dodavatelský řetězec (open-source kód) je primárním zdrojem kritického bezpečnostního dluhu. Modely AI jsou trénovány na stejném korpusu kódu. Modely se tedy učí z datové sady, o které je známo, že je plná bezpečnostních chyb. Výstup je statistickým odrazem vstupu. To vytváří zpětnovazební smyčku, kde nezabezpečený veřejný kód trénuje AI k produkci dalšího nezabezpečeného kódu, který může být nakonec publikován, což dále znečišťuje trénovací data pro budoucí modely. Jedná se o systémové, kumulativní riziko.

Část 4: Srovnávací analýza: Bezpečnostní stav kódu psaného člověkem vs. kódu generovaného AI

Tato část poskytuje přímé, na důkazech založené srovnání, které syntetizuje rozsáhlé akademické studie, jež staví lidský a AI kód proti sobě. Řeší zjevný rozpor mezi obecnými prohlášeními průmyslu a podrobným výzkumem.

4.1 Sjednocení dat: Počet chyb vs. závažnost zranitelností

Raný průmyslový výzkum naznačoval, že kód generovaný AI má "přibližně stejné procento bezpečnostních chyb jako kód generovaný lidmi".[8] Rozsáhlá akademická studie (přes 500 tisíc vzorků) z arXiv (2508.21634) však poskytuje nuancovanější obraz. Zatímco hrubý počet

defektů může být proměnlivý, **kód generovaný AI obsahuje více vysoce rizikových bezpečnostních zranitelností**.[2] Klíčový rozdíl spočívá mezi obecnými "defekty" (např. nepoužité proměnné, problémy se stylem) a bezpečnostními "zranitelnostmi" (např. command injection). AI může produkovat čistěji

vypadající kód s menším počtem nízkoúrovňových defektů, ale zároveň zavádět více kritických bezpečnostních děr.

4.2 Srovnávací analýza profilů defektů

Defekty v kódu generovaném AI jsou obvykle dominovány **problémy s přiřazením** (např. unused-argument, unused-variable) a používáním pevně zakódovaných ladicích konstrukcí (např. System.out.println()).[2] To ukazuje na kód, který je často naivní a postrádá připravenost pro produkční nasazení. Naopak, kód psaný člověkem vykazuje vyšší koncentraci

problémů s udržovatelností a algoritmických problémů, jako je vysoká cyklomatická složitost, nesprávné zpracování výjimek a problémy související s rozhraním (protected-access).[2] To odráží výzvy spojené se správou složitých, vyvíjejících se reálných kódových bází.

4.3 Srovnávací analýza bezpečnostních zranitelností (CWEs)

Klíčovým zjištěním akademické studie je, že napříč jazyky Python i Java byl lidmi psaný kód "konzistentně nejbezpečnější napříč všemi metrikami".[2]

Při srovnání v jazyce **Python** modely AI vygenerovaly **o 5 tisíc více zranitelných vzorků** než lidský základ. Kód AI neúměrně často spouštěl **CWE-78 (OS Command Injection)**, přičemž některé modely produkovaly více než 5x více instancí než lidský kód (13 419 vs. 2 243).[2]

V jazyce **Java** byl rozdíl ještě výraznější. Modely AI vygenerovaly **o 18 tisíc více zranitelných vzorků**. Jeden model AI (DeepSeek-Coder) produkoval více než 8x více instancí **CWE-532 (Information Exposure Through Log Files)** než lidský kód (30 031 vs. 3 736) a dominoval v **CWE-798 (Use of Hardcoded Credentials)**.[2]

4.4 Srovnávací analýza strukturální složitosti

Kód generovaný AI je obecně **jednodušší, kratší a více se opakující**. V průměru má méně řádků kódu (-6,75), nižší cyklomatickou složitost (-1,66) a výrazně méně unikátních tokenů, což naznačuje nižší lexikální rozmanitost.[2] Kód psaný člověkem naopak vykazuje

větší strukturální složitost, hlubší logické větvení a složitější řízení toku, což odráží návrh sofistikovanějších systémů.[2]

Kód generovaný AI a kód psaný člověkem selhávají zásadně odlišnými způsoby. Lidé zavádějí chyby prostřednictvím složitosti a vyvíjející se logiky; AI zavádí chyby prostřednictvím naivního porovnávání vzorů a neznalosti bezpečnostního kontextu. Akademická data jasně oddělují typy defektů. Hlavními defekty lidského kódu jsou protected-access, CyclomaticComplexity a AvoidCatchingGenericException – to jsou chyby návrhu, architektury a zkušeností. Hlavními defekty kódu AI jsou unused-argument a SystemPrintln – to jsou chyby naivity a nedostatku kontextu o účelu nebo prostředí kódu. Podobně data

o zranitelnostech ukazují, že lidé se potýkají s Improper Check for Unusual or Exceptional Conditions (CWE-754), což je chyba logické úplnosti. AI se potýká s OS Command Injection (CWE-78) a Hardcoded Credentials (CWE-798), což jsou selhání při aplikaci základních, univerzálních bezpečnostních principů. Bezpečnostní strategie proto musí být rozděleny. Pro lidský kód se zaměřte na revize návrhu a řízení složitosti. Pro kód AI se zaměřte na prosazování základní bezpečnostní hygieny a skenování s ohledem na kontext.

Jednoduchost kódu AI je dvousečná zbraň. I když se může zdát lépe udržovatelný, tato jednoduchost maskuje vyšší hustotu závažných bezpečnostních rizik. Metriky ukazují, že kód AI je méně složitý (méně řádků, nižší CCN), což by se tradičně považovalo za pozitivní vlastnost kvality. Stejné kódové báze však vykazují vyšší prevalenci kritických CWE. To znamená, že tradiční metriky kvality (jako je složitost) již nejsou spolehlivými zástupci bezpečnosti v éře AI. Jednoduchá, čistě vypadající funkce generovaná AI může být mnohem nebezpečnější než složitá, chaotická funkce napsaná bezpečnostně uvědomělým člověkem. To zneplatňuje staré předpoklady a vyžaduje, aby bezpečnostní týmy při hodnocení kódu generovaného AI upřednostňovaly přímé skenování zranitelností před nepřímými metrikami kvality.

Kategorie CWE	Lidský základ (% zranitelných vzorků)	Model AI 1 (ChatGPT) (% zranitelných vzorků)	Model AI 2 (DeepSeek-Coder) (% zranitelných vzorků)	Klíčové pozorování
Python				
CWE-78 (OS Command Injection)	2,243 instancí	13,419 instancí	10,908 instancí	AI modely generují dramaticky více zranitelností typu command injection.
CWE-754 (Improper Check)	13,157 instancí	N/A	N/A	Dominantní zranitelnost v lidském kódu, související s logickou komplexností.
Celkově zranitelné vzorky	5,55 %	~5,7 %	~5,7 %	AI produkuje více zranitelností s vyšší hustotou na vzorek.
Java				
CWE-532 (Info Exposure via Logs)	3,736 instancí	N/A	30,031 instancí	Model DSC produkuje masivní objem zranitelností úniku informací.

CWE-798 (Hardcoded Credentials)	N/A	N/A	Vysoká prevalence	AI modely jsou náchylné k vkládání citlivých údajů přímo do kódu.
Celkově zranitelné vzorky	3,00 %	8,18 %	19,56 %	Rozdíl v bezpečnosti je v Javě extrémně výrazný, AI kód je podstatně zranitelnější.

Tabulka 2: Srovnávací analýza zranitelností:

Kód psaný člověkem vs. kód generovaný AI (Python & Java) (Zdroj dat: [2])

Část 5: Paradox iterace: Jak "vylepšení" pomocí AI mohou zhoršit bezpečnost

Tato část se zaměřuje na kritické, protiintuitivní zjištění, které představuje zásadní změnu paradigmatu v chápání rizik spojených s AI.

5.1 Chybný předpoklad vylepšování

Běžným pracovním postupem je, že vývojáři vedou zpětnovazební smyčky s LLM, předkládají kód k vylepšení, upřesnění nebo rozšíření.[5] Implicitním předpokladem je, že tento proces zvyšuje kvalitu a bezpečnost kódu. Kontrolovaný experiment však tento předpoklad systematicky zpochybňuje. Studie (arXiv:2506.11022) začala s bezpečným výchozím kódem a podrobila ho několika kolům "vylepšení" pomocí AI. Výsledky ukazují, že tento proces může vést k významnému zhoršení bezpečnosti.[6]

5.2 Kvantifikace zhoršení bezpečnosti

Klíčovým statistickým údajem je, že studie zjistila **nárůst kritických zranitelností o 37,6 % již po pěti iteracích** "vylepšení" pomocí AI.[5] Vzor ukazuje jasný trend rostoucího rizika v čase:

- První iterace: průměrně 2,1 zranitelnosti na vzorek.
- Střední iterace (3-7): průměrně 4,7 zranitelnosti na vzorek.
- Pozdní iterace (8-10): průměrně 6,2 zranitelnosti na vzorek.[5]

5.3 Vliv strategie promptování

Experiment testoval čtyři odlišné strategie promptování: zaměřené na efektivitu ("optimalizuj toto"),

zaměřené na funkce ("přidej tuto funkci"), zaměřené na bezpečnost ("udělej toto bezpečnější") a nejednoznačné ("vylepši toto").[6] Na základě strategie promptování se objevily odlišné vzorce zranitelností, což naznačuje, že

záměr požadavku vývojáře ovlivňuje *typ* zavedené bezpečnostní chyby.[6]

Samostatný výzkum potvrzuje, že návrh promptu má hluboký dopad. Optimalizované, bezpečnostně uvědomělé prompty mohou snížit výskyt bezpečnostních rizik desetinásobně ve srovnání s neoptimalizovanými prompty. Avšak i s nejlepšími prompty se zranitelnosti stále objevují.[17]

Konverzační, iterativní povaha interakce s kódovacími AI je skrytou bezpečnostní hrozbou. Každá interakce je novou příležitostí k zavedení chyby a model nemá žádnou trvalou "paměť" počátečního bezpečného stavu. Vývojář začne s bezpečným kusem kódu. Požádá AI, aby ho "udělala efektivnějším". AI, optimalizující výkon, může odstranit ověřovací kontrolu, kterou považuje za výpočetně náročnou, a tím zavést zranitelnost. Vývojář poté požádá o "přidání logování". AI, zaměřená na tento nový úkol, může logovat citlivá data a zavést další zranitelnost. Model zachází s každým promptem jako s diskrétním úkolem bez kontextu. Neprovádí holistickou bezpečnostní regresní analýzu svých vlastních změn. To znamená, že běžný pracovní postup "párového programátora" je zásadně nebezpečný. Nejedná se o spolupráci mezi rovnými, ale o sérii instrukcí daných nástroji, který může tiše porušit kritická bezpečnostní omezení s každým krokem.

Prompty zaměřené na bezpečnost nejsou všelékem a mohou být zavádějící. I když mohou opravit jeden problém, mohou snadno zavést jiný, což vytváří scénář "mlácení krtků" se zranitelnostmi. Studie zahrnovala strategii promptování zaměřenou na bezpečnost. Navzdory tomu přetrvával celkový trend zhoršování bezpečnosti v průběhu iterací. To naznačuje, že i když je LLM explicitně řečeno, aby se zaměřil na bezpečnost, jeho změny nejsou zaručeně holisticky bezpečné. Může opravit známou chybu XSS, ale v procesu zavést nový problém s kontrolou přístupu. To podkopává myšlenku, že bezpečnost lze do kódu "napromptovat" a posiluje potřebu externího, nezávislého ověření po *každé* významné změně řízené AI.

Číslo iterace	Průměrný počet zranitelností na vzorek	% nárůst kritických zranitelností od základu
Iterace 1	2,1	N/A
Iterace 5	N/A	+37,6 %
Iterace 10	6,2	N/A

Tabulka 3: Vývoj počtu zranitelností v kódu AI prostřednictvím iterativního vylepšování (Zdroj dat: [5])

Část 6: Strategické důsledky a doporučení pro

budoucnost hybridního vývoje

Tato závěrečná část převádí předchozí analýzu do souboru akčních strategií pro CISO a vedoucí inženýrských týmů, aby se mohli orientovat v rizicích a příležitostech vývoje s podporou AI.

6.1 Vývoj bezpečného životního cyklu vývoje softwaru (SSDLC) pro éru AI

Základní princip se musí posunout z "Shift Left" na "Shift Everywhere with Skeptical Oversight". Rychlost a rozsah AI znamenají, že bezpečnost nemůže být jednofázovou bránou, ale nepřetržitým, integrovaným procesem, který zachází s kódem generovaným AI jako s inherentně nedůvěryhodným.[18]

Veškerý netriviální kód generovaný nebo upravený AI musí projít povinnou revizí bezpečného kódu provedenou lidským vývojářem vyškoleným k odhalování specifických vzorců chyb AI (např. architektonický posun, chyby bez kontextu).[24] Toto je nejdůležitější kontrolní mechanismus.

Nástroje musí být přizpůsobeny:

- **SAST (Static Application Security Testing):** Nástroje musí být vyladěny nebo vylepšeny, aby specificky detekovaly vysokofrekvenční zranitelnosti zaváděné AI (CWE-78, CWE-798 atd.).
- **SCA (Software Composition Analysis):** Nástroje musí být integrovány s mechanismy pro detekci a označení "halucinovaných" závislostí před jejich instalací.
- **DAST (Dynamic Application Security Testing):** Dynamické testování se stává ještě kritičtějším pro odhalení logických a architektonických chyb, které by SAST v kódu AI mohl přehlédnout.

6.2 Kritická role podpory vývojářů a řízení

Organizace musí vyvinout a prosazovat pokyny pro bezpečný prompt engineering. Školení vývojářů v tvorbě promptů, které jsou explicitní ohledně bezpečnostních omezení, zpracování dat a chybových stavů, je vysoce účinnou a nízkonákladovou intervencí.[17]

Rychlost AI vytváří tlak na obcházení bezpečnosti. 80 % vývojářů přiznává, že obchází bezpečnostní politiky AI, když jsou pod tlakem.[4] Bezpečnostní nástroje musí být bezproblémově integrovány do IDE a CI/CD pipeline, aby poskytovaly okamžitou, akční a bezproblémovou zpětnou vazbu, čímž se bezpečná cesta stane cestou nejmenšího odporu.[28]

Rychlost generování kódu pomocí AI znamená, že hustota zranitelností a bezpečnostní dluh se mohou hromadit mnohem rychleji. Rizikové modely musí být aktualizovány, aby zohledňovaly tento zrychlený rizikový profil. Doba nápravy musí být drasticky zkrácena pro aplikace s významným podílem kódu generovaného AI.[29]

6.3 Rámec pro bezpečné přijetí AI

Je nutné implementovat politiku založenou na riziku. Kód generovaný AI pro necitlivé interní nástroje

může vyžadovat peer review, zatímco kód pro kritické, internetově dostupné aplikace s citlivými daty vyžaduje revizi seniorním vývojářem nebo bezpečnostním šampionem.

Je třeba zavést pevnou politiku, že jakmile je kus kódu schválen, nemůže být iterativně "vylepšován" AI bez opětovného zahájení celého procesu revize bezpečného kódu od začátku.

Krajina schopností a rizik AI se rychle vyvíjí. Program neustálého vzdělávání pro vývojáře i bezpečnostní týmy je nezbytný pro udržení kroku s novými hrozbami a technikami zmírňování.

Tradiční model SSDLC, který předpokládá lineární a lidským tempem řízený tok, je narušen rychlostí a iterativní povahou AI. Je zapotřebí nový, dynamičtější a skeptičtější model. SSDLC je navržen tak, aby přidával bezpečnostní kontroly v diskrétních fázích (návrh, kód, test, nasazení). AI umožňuje vývojáři projít těmito fázemi pro jednu funkci během několika minut prostřednictvím iterativního promptování. Jak bylo ukázáno, tato rychlá iterace zavádí zranitelnosti. Bezpečnostní model tedy již nemůže být sérií bran. Musí to být nepřetržitý obal skepticismu kolem interakce vývojáře a AI, se silným důrazem na lidskou revizi jako konečného arbitra před jakýmkoli commitem kódu.

Primární výzvou AI v softwarové bezpečnosti není technická, ale sociálně-technická. Jedná se o konflikt mezi uživatelským zážitkem vývojáře s AI (rychlý, snadný, bezproblémový) a uživatelským zážitkem vývojáře s tradiční bezpečností (pomalý, obtížný, plný překážek). Vývojáři jsou motivováni k rychlému dodávání funkcí a AI to dramaticky zrychluje. Bezpečnostní revize a náprava zranitelností jsou zdrojem tření, které tento proces zpomaluje. Data ukazují, že vývojáři budou obcházet bezpečnostní kontroly, aby si udrželi rychlost. Vítěznou strategií tedy není přidávat další brány (které budou obcházeny), ale snižovat tření spojené s bezpečnými postupy. To znamená investovat do nástrojů, které poskytují okamžitou, přesnou a užitečnou zpětnou vazbu přímo v pracovním postupu vývojáře, aby se bezpečnost stala součástí zrychlení řízeného AI, nikoli překážkou.

Citovaná díla

1. 2025 State of Software Security - Veracode, použito září 9, 2025, https://www.veracode.com/wp-content/uploads/2025/02/State-of-Software-Security-2025.pdf
2. Human-Written vs. AI-Generated Code: A Large-Scale Study ... - arXiv, použito září 9, 2025, https://arxiv.org/pdf/2508.21634
3. AI can write your code, but nearly half of it may be insecure - Help Net Security, použito září 9, 2025, https://www.helpnetsecurity.com/2025/08/07/create-ai-code-security-risks/
4. Is Vibe Coding Putting Us All at Risk? - C# Corner, použito září 9, 2025, https://www.c-sharpcorner.com/article/is-vibe-coding-putting-us-all-at-risk/
5. Security Degradation in Iterative AI Code Generation: A Systematic Analysis of the Paradox, použito září 9, 2025, https://arxiv.org/html/2506.11022v1
6. Security Degradation in Iterative AI Code Generation -- A Systematic Analysis of the Paradox, použito září 9, 2025, https://www.researchgate.net/publication/392716752_Security_Degradation_in_Iterative_AI_Code_Generation_--_A_Systematic_Analysis_of_the_Paradox
7. Security Degradation in Iterative AI Code Generation: A ... - arXiv, použito září 9, 2025, https://arxiv.org/pdf/2506.11022
8. State of Software Security 2024 - Veracode, použito září 9, 2025, https://www.veracode.com/wp-content/uploads/2024/06/SOSS-Report-2024.pdf
9. OWASP Top Ten | OWASP Foundation, použito září 9, 2025, https://owasp.org/www-project-top-ten/
10. The OWASP Top 10 Explained: Today's Top Risks in Web Apps and LLMs | Splunk, použito září 9, 2025, https://www.splunk.com/en_us/blog/learn/owasp-top-10.html
11. The OWASP Top Ten 2025, použito září 9, 2025, https://www.owasptopten.org/
12. State of Software Security 2024 Report | Veracode, použito září 9, 2025, https://www.veracode.com/state-software-security-2024-report/
13. State of Software Security (SoSS) 2025: A New View of Maturity | Veracode, použito září 9, 2025,

https://www.veracode.com/resources/analyst-reports/state-of-software-security-2025/
14. State of Open Source security 2022 | Snyk, použito září 9, 2025, https://snyk.io/reports/open-source-security-2022/
15. The Most Common Security Vulnerabilities in AI-Generated Code ..., použito září 9, 2025, https://www.endorlabs.com/learn/the-most-common-security-vulnerabilities-in-ai-generated-code
16. The Impact of AI-Assisted Code Generation on Software Vulnerabilities and the Role of AI in Automated Security Testing - ResearchGate, použito září 9, 2025, https://www.researchgate.net/publication/391706149_The_Impact_of_AI-Assisted_Code_Generation_on_Software_Vulnerabilities_and_the_Role_of_AI_in_Automated_Security_Testing
17. Security Flaws In Generative AI Code - DiVA, použito září 9, 2025, https://kth.diva-portal.org/smash/get/diva2:1985840/FULLTEXT01.pdf
18. What is Secure SDLC (SSDLC)? - Wiz, použito září 9, 2025, https://www.wiz.io/academy/secure-sdlc
19. All about the Secure Software Development Lifecycle (SSDLC) - Codecademy, použito září 9, 2025, https://www.codecademy.com/article/all-about-the-secure-software-development-lifecycle-ssdlc
20. Mastering DAST vs. SAST: An Ultra-Extensive Guide to Application Security Testing, použito září 9, 2025, https://medium.com/@okanyildiz1994/mastering-dast-vs-sast-an-ultra-extensive-guide-to-application-security-testing-aadcc6c26478
21. SDLC Security: Everything You Need to Know, použito září 9, 2025, https://www.ox.security/blog/sdlc-security-everything-you-need-to-know/
22. What is a Secure Software Development Lifecycle (SSDLC)? - JFrog, použito září 9, 2025, https://jfrog.com/learn/devsecops/ssdlc-secure-software-development-lifecycle/
23. How To Incorporate SAST, DAST, And SCA Into The SDLC - Checkmarx, použito září 9, 2025, https://checkmarx.com/learn/appsec/incorporate-sast-sca-dast-in-sdlc/
24. Best practices for secure code review, použito září 9, 2025, https://blog.bytehackr.in/5-best-practices-for-secure-code-review-strategies-examples-and-tools
25. What is a Secure Code Review? - CloudDefense.AI, použito září 9, 2025, https://www.clouddefense.ai/what-is-a-secure-code-review/
26. OWASP Code Review Guide v2-1-10 | PDF | Source Code | Unit ..., použito září 9, 2025, https://www.scribd.com/document/726763869/OWASP-Code-Review-Guide-v2-1-10
27. Secure Code Review | PDF | Computing - Scribd, použito září 9, 2025, https://www.scribd.com/document/893321343/Secure-Code-Review
28. Benefits of a Developer-First Approach: Enhance Security & Innovation, použito září 9, 2025, https://boostsecurity.io/blog/benefits-developer-first-approach?hsLang=en
29. Beyond Shift-Left: Rethinking AppSec Strategies in the Age of AI - Jit.io, použito září 9, 2025, https://www.jit.io/resources/devsecops/beyond-shift-left-rethinking-appsec-strategies-in-the-age-of-ai