# Sistemski napadi na programske dobavne verige v dobi umetne inteligence

Gregor Jošt, Viktor Taneski

Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija
gregor.jost@um.si, viktor.taneski@um.si

Programske dobavne verige so zaradi vse večje odvisnosti od odprtokodnih komponent, avtomatiziranih razvojnih procesov, oblačnih storitev in medsebojno povezanih identitetnih sistemov postale pomembna tarča kibernetskih napadov. Prispevek analizira njihov razvoj od zlorab posameznih paketov in računov vzdrževalcev do večstopenjskih sistemskih napadov, ki izkoriščajo povezave zaupanja med repozitoriji, okolji CI/CD, registri artefaktov, kontejnerskimi slikami in produkcijsko infrastrukturo. Obravnavani so ključni napadalni vektorji, vključno z zastrupljanjem paketov, zlorabo razvojnih identitet, krajo skrivnosti, napačnimi nastavitvami OIDC in zlorabo procesov gradnje ter distribucije. Dodatna pozornost je namenjena umetni inteligenci, ki sama po sebi ni nov vzrok napadov, vendar povečuje hitrost razvoja, obseg avtomatizacije in količino sprememb, ki zahtevajo preverjanje. Razvojni agenti dodatno širijo napadalno površino zaradi dostopa do kode, orodij in privilegiranih identitet. Prispevek ugotavlja, da učinkovita zaščita zahteva sistemski in večplastni pristop, ki združuje upravljanje odvisnosti, preverljivo poreklo artefaktov, utrjevanje okolij CI/CD, najmanjše privilegije, nadzor dobaviteljev ter neprekinjeno spremljanje celotne verige zaupanja.

**Ključne besede:**
programske dobavne verige
napadi na dobavne verige
verige zaupanja
programske odvisnosti
generativna umetna inteligenca

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-299-188-3
DOI https://doi.org/10.18690/um.feri.7.2026.11

# 1 Uvod

Programska oprema danes praviloma ni rezultat izoliranega razvoja v eni organizaciji, temveč nastaja v razvejanem ekosistemu odprtokodnih knjižnic, registrov paketov, sistemov za upravljanje izvorne kode, okolij in storitev za neprekinjeno integracijo in dostavo (angl. Continuous Integration and Continuous Delivery, CI/CD), oblačne infrastrukture ter zunanjih ponudnikov. Programsko dobavno verigo zato razumemo kot celoto ljudi, procesov, orodij in artefaktov, ki sodelujejo pri načrtovanju, razvoju, gradnji, testiranju, distribuciji, uvajanju in vzdrževanju programske rešitve. Varnost končnega izdelka ni odvisna zgolj od kakovosti lastne izvorne kode, temveč tudi od integritete vseh komponent, orodij in odnosov zaupanja v celotni verigi izdelave in distribucije programske opreme [1], [2].

Takšna povezanost prinaša pomembne razvojne prednosti, hkrati pa ustvarja izrazit sistemski učinek. Napadalcu ni treba neposredno ogroziti vsake ciljne organizacije, saj je učinkoviteje prevzeti komponento, račun, storitev ali avtomatizirani proces, ki mu zaupa več uporabnikov. Agencija Evropske unije za varnost omrežij in informacij (ENISA) napad na dobavno verigo opredeljuje kot napad, pri katerem napadalec za dosego končnega cilja napade dobavitelja in izkoristi zaupanje med dobaviteljem in njegovimi strankami. Analiza incidentov kaže, da tarča ni nujno samo koda, temveč tudi mehanizmi posodabljanja, okolja za gradnjo programske opreme (angl. build environment), podpisni ključi, identitete delovnih procesov in distribucijski kanali [2]. Posledično lahko en sam varnostni incident povzroči kaskadno širjenje zlonamerne kode ali nepooblaščenega dostopa med velikim številom organizacij.

Raziskovalni problem tega prispevka izhaja iz dejstva, da klasične obravnave napadov na dobavne verige pogosto analizirajo posamezne vektorje ločeno, na primer zlonamerne pakete, prevzete račune vzdrževalcev ali zlorabljena okolja za gradnjo programske opreme. Sodobni incidenti pa kažejo, da napadalci te vektorje vse pogosteje povezujejo v večstopenjske operacije: začetna zloraba odvisnosti ali razvijalca omogoči dostop do okolja CI/CD, tam pridobljene skrivnosti in identitete pa se uporabijo za prevzem dodatnih repozitorijev, registrov, oblačnih storitev ali produkcijskih sistemov. Posledično je osrednje vprašanje prispevka, kako se napadi razvijajo od zlorabe posamezne komponente v sistemske napade na verige zaupanja in kako generativna umetna inteligenca spreminja hitrost, obseg in nadzor nad takšnimi procesi.

Cilji prispevka so: (1) prikazati zgodovinski razvoj napadov na programske dobavne verige; (2) sistematizirati ključne napadalne vektorje sodobnih razvojnih ekosistemov; (3) pojasniti njihovo povezovanje v večstopenjske in kaskadne napadalne verige; (4) analizirati vlogo umetne inteligence kot pospeševalca razvoja in napadov; ter (5) predstaviti tehnične in organizacijske obrambne pristope. Prispevek obstoječo literaturo dopolnjuje s sistemskim pogledom, v katerem programska dobavna veriga ni predstavljena kot zaporedje neodvisnih komponent, temveč kot dinamično omrežje identitet, avtomatiziranih procesov in prenosljivega zaupanja. Takšen pogled je skladen tudi s sodobnimi okviri, kot sta NIST Secure Software Development Framework (v nadaljevanju SSDF) in Supply-chain Levels for Software Artifacts (v nadaljevanju SLSA), ki varnosti ne omejujeta na pregled izvorne kode, temveč poudarjata zaščito razvojnega okolja, integriteto artefaktov in preverljivo poreklo programske opreme [1][3].

# 2 Evolucija napadov na programske dobavne verige

Zamisel, da je mogoče ogroziti program prek orodij in procesov, s katerimi je bil ustvarjen, ni nova. Thompson je že leta 1984 v delu Reflections on Trusting Trust pokazal, da je lahko zlonamerna funkcionalnost skrita v prevajalniku in se ohranja tudi takrat, ko je izvorna koda ciljnega programa navidezno neoporečna [4]. Temeljna ugotovitev je ostala aktualna: preverjanje posamezne kode ne zadošča, če ni mogoče zaupati orodju, okolju in postopku, ki so to kodo pretvorili v izvedljiv artefakt.

Z razmahom odprtokodne programske opreme in javnih registrov paketov se je problem zaupanja razširil na množico ponovno uporabljenih komponent. Razvijalci so začeli aplikacije sestavljati iz neposrednih in prehodnih

programskih odvisnosti (angl. software dependencies), ki jih pogosto vzdržujejo posamezniki ali majhne skupnosti. Napadalci so to okolje izkoristili z objavo zlonamernih paketov, zamenjavo podobno poimenovanih paketov, prevzemom zapuščenih projektov in zlorabo računov vzdrževalcev. Ohm in sodelavci so ročno zbrali in analizirali 174 zlonamernih paketov, ki so bili med novembrom 2015 in novembrom 2019 uporabljeni v dejanskih napadih ter distribuirani prek registrov npm, PyPI in RubyGems. Analiza je pokazala, da so bili takšni napadi že pred velikimi sistemskimi incidenti raznoliki, praktično izvedljivi in pogosto dolgo neodkriti [5].

Naslednja razvojna stopnja je bila usmeritev v infrastrukturo za gradnjo in distribucijo. Incidenti, kot sta SolarWinds in Codecov, so pokazali, da lahko zloraba zaupanja vrednega postopka posodabljanja ali razvojne storitve napadalcu omogoči dostop do številnih uporabnikov brez neposrednega napada na njihove sisteme. Pri SolarWinds je bila zlonamerna koda vključena v legitimne posodobitve platforme Orion, pri Codecovu pa je napadalec spremenil Bash Uploader in prek izvajanja v okoljih CI pridobival okoljske spremenljivke ter druge občutljive podatke [6], [7]. S tem se je težišče premaknilo od vprašanja, ali je ranljiva posamezna knjižnica, k vprašanju, kdo nadzira proces, ki programsko opremo gradi, podpisuje in objavlja.

Današnji napadi nadaljujejo ta trend z izkoriščanjem povezav med registri paketov, repozitoriji, CI/CD platformami, storitvami SaaS, kontejnerskimi registri, oblačnimi računi in federiranimi identitetami. Programska dobavna veriga je postala omrežje, v katerem legitimna poverilnica (angl. credential) ali zaupanja vredna avtomatizacija pogosto omogoča večji doseg kot tehnična ranljivost v končni aplikaciji. Evolucijo napadov zato lahko razumemo kot prehod od vstavljanja zlonamerne kode v posamezen artefakt k sistematičnemu prevzemanju mehanizmov, ki vzpostavljajo in širijo zaupanje.

### 2.1. Tradicionalni modeli napadov

Tradicionalni model napada je bil praviloma osredotočen na eno komponento ali eno distribucijsko točko. Napadalec je ustvaril zlonameren paket, posnemal ime legitimne knjižnice, prevzel račun vzdrževalca ali v obstoječo odvisnost vključil zlonamerno funkcionalnost. Uspeh je bil odvisen od tega, ali bodo razvijalci paket uporabili kot neposredno odvisnost ali pa bo do njihovih projektov prišel prek prehodne odvisnosti (angl. transitive dependency). Napadi so pogosto uporabljali namestitvene skripte, prikrito omrežno komunikacijo, krajo lokalnih poverilnic ali ciljno logiko, ki se je aktivirala samo v določenem okolju [5][8].

Kljub velikemu potencialnemu dosegu so bili zgodnji modeli pogosto omejeni glede nadaljnjega širjenja. Zlonamerna koda je bila vezana na konkretni paket, operacijski sistem ali skupino uporabnikov; napadalec je moral čakati na namestitev, posodobitev ali napako razvijalca. Prav tako je bila povezava med začetno zlorabo in drugimi deli razvojne infrastrukture manj avtomatizirana. Napad je lahko ukradel lokalne podatke ali vzpostavil začetni dostop, ni pa nujno samodejno ustvaril novih zaupanja vrednih objav, prevzel drugih paketov ali pridobil identitet za dostop do oblačne infrastrukture.

Te omejitve ne pomenijo, da so tradicionalni napadi izginili. Nasprotno, še vedno predstavljajo pogost začetni vektor, vendar so danes pogosto samo prva faza širše operacije. Njihova nevarnost se bistveno poveča, ko je okužena komponenta izvedena v okolju CI/CD z razširjenimi pravicami, kjer so dostopni žetoni (angl. access tokens) za objavo paketov, ključi za oblak, podpisne poverilnice ali pravice za spreminjanje drugih repozitorijev.

### 2.2. Prehod v sistemske napade

Prehod v sistemske napade je posledica spremembe arhitekture razvoja programske opreme. Sodobni razvojni tokovi avtomatizirano povezujejo spremembo izvorne kode z gradnjo, testiranjem, varnostnim preverjanjem, podpisovanjem, objavo artefakta in namestitvijo v produkcijo. Vsaka faza uporablja storitvene račune, žetone, skrivnosti ali federirane identitete (angl. federated identities; identitete, ki omogočajo dostop do več povezanih storitev brez ločenih računov), zato začetna zloraba ne predstavlja več nujno končnega cilja, temveč izhodišče za premikanje po celotni verigi.

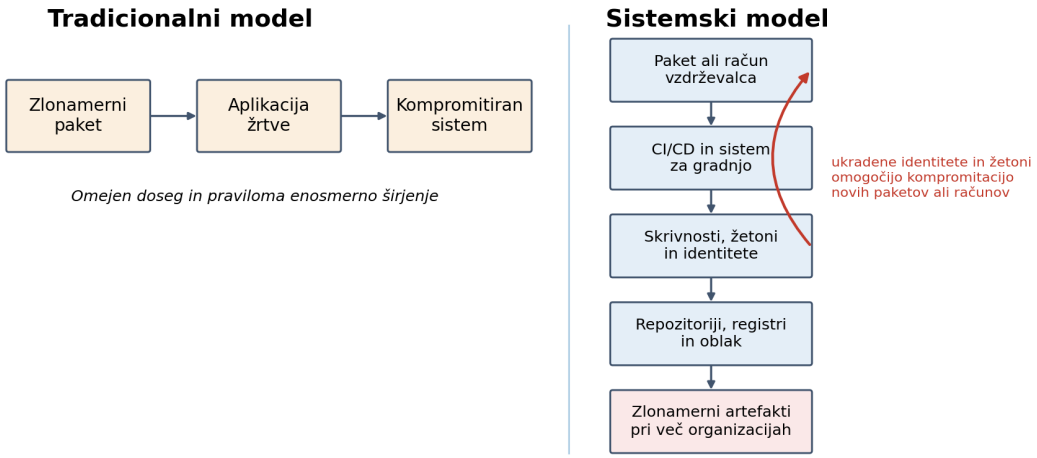
Okolje CI/CD je pri tem posebej pomembno, ker združuje izvorno kodo, orodja za gradnjo in visoko privilegirane dostopne podatke. Spremenjen delovni tok ali proces gradnje lahko prebere okoljske spremenljivke (angl.

environment variables), zlorabi izvajalno okolje, spremeni končni artefakt ali objavi novo različico prek legitimnega računa. Izrazit primer takšnega sistemskega napada je zloraba XZ Utils oziroma knjižnice liblzma, odkrita marca 2024. Napadalec je z večletnim socialnim inženiringom postopoma pridobil položaj zaupanja vrednega vzdrževalca projekta, nato pa je v izdaji 5.6.0 in 5.6.1 vključil prikrito zlonamerno logiko. Ključni deli mehanizma niso bili neposredno vidni v običajni izvorni kodi, temveč so bili vključeni v distribucijske arhive in aktivirani med procesom gradnje, kar je v določenih okoljih omogočilo poseganje v avtentikacijsko pot strežnika OpenSSH [9], [10].

Oblačne in SaaS storitve dodatno razširjajo napadalno površino. Repozitorij kode, CI/CD platforma, register paketov in oblačni ponudnik so lahko ločene storitve, vendar jih povezujejo enotna prijava (angl. Single Sign-On, SSO), aplikacijske integracije, dogodkovni klici HTTP in avtomatizirane vloge. Zloraba identitete v eni storitvi lahko zato odpre pot v druge sisteme. Posebej občutljive so preširoko nastavljene federirane povezave OIDC: če napadalec doseže izvajanje kode v zaupanja vrednem delovnem toku, lahko pridobi veljaven kratkoživ žeton in ga uporabi za dostop do oblačnih virov. V tem primeru ni zaobšel avtentikacije, temveč je zlorabil pravilo zaupanja, ki je njegovo dejavnost prepoznalo kot legitimno [11].

Sistemske napade dodatno zaznamujeta avtomatizacija in sposobnost samodejnega širjenja. Ukradeni žetoni za objavo paketov se lahko uporabijo za zlorabo novih paketov, ti pa v okoljih naslednjih žrtev pridobijo nove poverilnice. Nastane povratna zanka, v kateri vsak uspešen napad poveča napadalčev doseg in število zaupanja vrednih identitet. Sodobne kampanje zato niso več omejene na en artefakt, temveč lahko sočasno ciljajo razvijalske delovne postaje, repozitorije, CI/CD izvajalce (angl. runners), registre in oblačne račune.

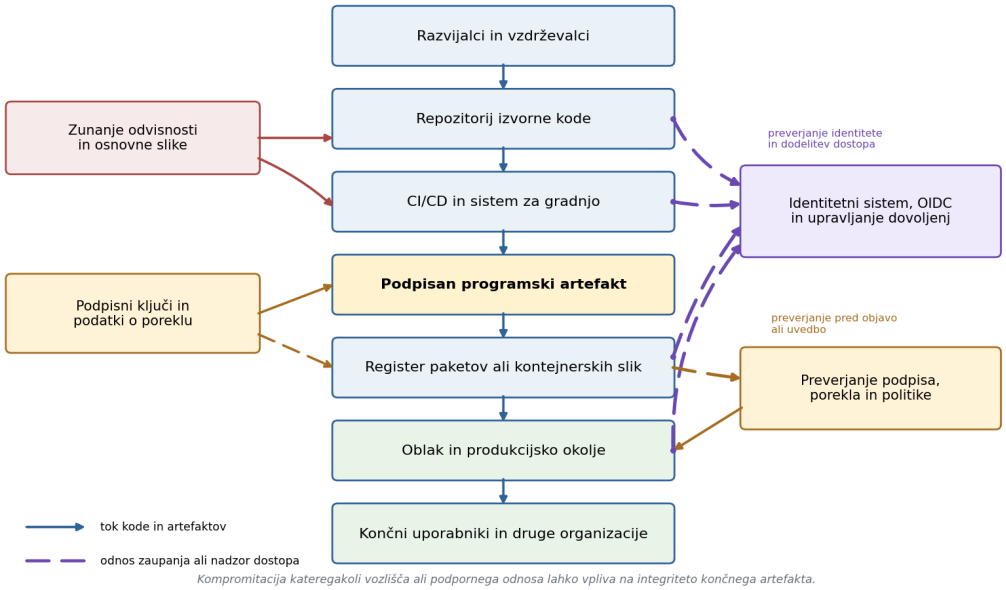
Kot prikazuje slika 1, je ključna sprememba v razmišljanju napadalcev premik od vprašanja »v katero komponento je mogoče vnesti zlonamerno spremembo« k vprašanju »katera povezava zaupanja omogoča napadalcu največji nadaljnji doseg«. V sistemskem modelu so koda, identiteta, avtomatizacija in infrastruktura medsebojno zamenljive poti do istega cilja. Obramba mora zato zagotavljati preverljivo sled izvora in gradnje, ločevanje privilegijev, kratkožive in strogo omejene identitete ter neodvisno preverjanje artefaktov. NIST SSDF in SLSA takšno usmeritev formalizirata z zahtevami po zaščitenih razvojnih okoljih, nadzoru sprememb, integriteti gradnje in podatkih o poreklu (angl. provenance) [1], [3].




**Slika 1: Prehod od kompromitacije posamezne komponente k sistemskemu napadu na verigo zaupanja.**

## 3 Ključni napadalni vektorji sodobnih dobavnih verig

Kot je bilo že izpostavljeno, varnost programske dobavne verige ni odvisna zgolj od posameznih komponent, temveč od povezav med sistemi, identitetami in avtomatiziranimi procesi, ki omogočajo njihov nastanek in distribucijo. Slika 2 prikazuje glavni tok programske opreme od razvijalcev in izvorne kode prek gradnje, podpisovanja in objave do produkcije ter končnih uporabnikov.




**Slika 2: Tok programske opreme in podporni odnosi zaupanja v sodobni programski dobavni verigi.**

V nadaljevanju so predstavljeni ključni napadalni vektorji sodobnih programskih dobavnih verig, izbrani glede na njihovo povezanost z osrednjimi točkami zaupanja v procesu razvoja in distribucije programske opreme. Obravnavani so napadi na programske odvisnosti (angl. software dependencies), zlorabe računov razvijalcev in vzdrževalcev (angl. maintainer account compromise), napadi na okolja CI/CD ter zlorabe identitetnih sistemov in federiranega dostopa (angl. federated identity and access). Poseben poudarek je namenjen točkam zaupanja, katerih zloraba omogoča napadalcem širjenje vpliva prek več komponent in organizacij. Tabela 1 povzema obravnavane vektorje, pripadajoče točke zaupanja in značilne posledice, ki jih podrobneje obravnavajo naslednja podpoglavja.

**Tabela 1:** Pregled ključnih napadalnih vektorjev in kompromitiranih točk zaupanja.

| Napadalni vektor | Točka zaupanja | Možna posledica | Primer |
|---|---|---|---|
| Zmeda odvisnosti in zastrupljanje paketov | Register in izbira odvisnosti | Izvajanje zlonamernega paketa | Birsanova raziskava; event-stream |
| Prevzem računa ali vzdrževanja | Identiteta razvijalca oziroma vzdrževalca | Legitimna objava škodljive različice | event-stream; ua-parser-js |
| Zloraba CI/CD | Delovni tokovi, izvajalci in skrivnosti | Kraja poverilnic ter dostop do drugih sistemov | Codecov; Trivy; KICS |
| Napačna nastavitev OIDC | Pravila federiranega zaupanja | Pridobitev legitimnih začasnih poverilnic | TanStack |
| Kompromitirana kontejnerska slika | Register in proces gradnje slike | Širjenje škodljive kode v razvoj in produkcijo | Docker Hub kampanje |

Tabela prikazuje napadalne vektorje ločeno zaradi lažje analize posameznih točk zaupanja, vendar v praksi ti vektorji niso nujno neodvisni. Sodobni napadi na programske dobavne verige pogosto kombinirajo več tehnik, pri čemer kompromitacija ene komponente omogoči dostop do naslednje. Posamezen incident lahko zato vključuje več navedenih kategorij hkrati: napadalec lahko na primer z zlorabo CI/CD procesa pridobi dostopne žetone, jih

uporabi za objavo zlonamernega paketa in nato izkoristi zaupanje uporabnikov v legitimne posodobitve. Takšne večstopenjske napadalne verige so podrobneje obravnavane v naslednjem poglavju.

### 3.1. Zmeda odvisnosti in zastrupljanje paketov

Razvijalci pri izdelavi aplikacij redno uporabljajo odprtokodne knjižnice in pakete, ki jih vključijo v svoje projekte prek upravljalnikov paketov (angl. package managers), kot so npm za JavaScript, PyPI za Python ali Maven za Javo. Tak pristop omogoča hitrejši razvoj, vendar hkrati pomeni, da organizacija zaupa tudi kodi, ki je sama neposredno ne razvija in ne nadzoruje. Programske odvisnosti (angl. software dependencies) zato predstavljajo pomemben del dobavne verige programske opreme. Čeprav razvijalci pogosto preverjajo lastno kodo, se lahko zlonamerna funkcionalnost skriva v eni izmed uporabljenih zunanjih komponent. Napadalec tako ne napade neposredno končnega sistema, temveč izkoristi zaupanje, ki ga organizacija namenja svojim odvisnostim.

Zmeda odvisnosti (angl. dependency confusion) je napad, pri katerem napadalec izkoristi način, kako sistemi za upravljanje paketov izbirajo in prenašajo odvisnosti. V številnih organizacijah obstajajo zasebni paketi (angl. private packages), ki niso javno objavljeni, vendar jih razvojna okolja uporabljajo pri gradnji aplikacij. Napadalec lahko ustvari javni paket z enakim imenom in ga objavi v javnem registru, na primer npm ali PyPI. Če sistem za namestitev paketov ne razlikuje pravilno med zasebnim in javnim virom, lahko prenese napadalčev paket namesto notranje različice. S tem napadalec vstavi svojo kodo neposredno v razvojni proces organizacije [12]. Koncept zmede odvisnosti je leta 2021 javno predstavil Alex Birsan, ki je pokazal, da je mogoče s to tehniko vplivati na razvojna okolja številnih velikih podjetij, med drugim Microsofta, Appla in drugih organizacij. Njegova raziskava je pokazala, da lahko že sama kombinacija javnega registra paketov in nepravilno konfiguriranih odvisnosti predstavlja pomembno varnostno tveganje [13].

Zastrupljanje paketov (angl. package poisoning) predstavlja širši razred napadov, pri katerih napadalec v programski paket vstavi zlonamerno kodo. Paket je lahko povsem nov ali pa gre za obstoječ paket, ki ga razvijalci že uporabljajo in mu zato zaupajo. Cilj napadalca je pogosto pridobiti dostop do razvojnih okolij, ukrasti skrivnosti (angl. secrets) ali izvesti nadaljnje napade v organizaciji [14]. Poseben problem predstavlja dejstvo, da imajo številni odprtokodni paketi veliko število uporabnikov, vendar jih vzdržuje majhno število posameznikov. Zaradi tega lahko zloraba ene same komponente vpliva na veliko število aplikacij in organizacij.

Eden izmed najbolj znanih primerov je incident s paketom event-stream v ekosistemu npm leta 2018. Napadalec je pridobil dostop do vzdrževanja priljubljenega paketa in vanj vključil dodatno odvisnost, ki je vsebovala zlonamerno kodo. Ker je bil paket že uveljavljen in široko uporabljen, je napad dosegel veliko število uporabnikov [15], [16]. Podoben primer predstavlja kompromitacija paketa ua-parser-js leta 2021, pri katerem je bila v uveljavljeni paket npm vključena zlonamerna koda. Napad je pokazal, da lahko zlonamerna sprememba enega samega priljubljenega paketa vpliva na veliko število razvojnih okolij in aplikacij [15].

Napadalci pogosto izkoriščajo tudi človeške napake pri izbiri paketov. Zloraba tipkarskih napak (angl. typosquatting) je tehnika, pri kateri napadalec ustvari paket z imenom, ki je zelo podobno priljubljenemu paketu. Primer takega pristopa je ustvarjanje paketov z imeni, pri katerih so spremenjene posamezne črke, dodani dodatni znaki ali uporabljene različice, podobne izvirnemu imenu. Ker razvijalci pogosto iščejo pakete neposredno prek registrov, lahko takšni paketi na prvi pogled delujejo verodostojno, dokler uporabnik ne preveri njihovega izvora [17]. Pri klasičnih spletnih napadih se typosquatting pogosto povezuje z lažnimi spletnimi stranmi, vendar enak princip velja tudi za ekosisteme programskih paketov, kot so npm, PyPI in drugi registri.

Poleg ustvarjanja novih zlonamernih paketov napadalci izkoriščajo tudi obstoječe projekte. Zapuščeni paketi (angl. abandoned packages) so knjižnice, ki jih razvijalci sicer še vedno uporabljajo, vendar jih prvotni vzdrževalci ne posodabljajo več aktivno. Če napadalec pridobi nadzor nad takim paketom, ga lahko spremeni in objavi novo različico, ki jo uporabniki samodejno prenesejo kot del običajnega procesa posodabljanja. Problem je še posebej izrazit pri ekosistemih, kjer se zaupa predvsem imenu paketa in številki različice, manj pa dejanskemu preverjanju sprememb.

Napadi na programske pakete kažejo, da dobavna veriga temelji na številnih povezavah zaupanja. Napadalci izkoriščajo slabosti registrov, paketov, človeških napak in vzdrževanja odprtokodnih projektov, pri čemer za širjenje vpliva pogosto zadošča ena sama ogrožena komponenta.

### 3.2. Kompromitacija računov vzdrževalcev in razvijalcev

Razvijalci in vzdrževalci odprtokodnih projektov imajo pogosto posebno vlogo, saj njihov račun omogoča objavo novih različic paketov, spreminjanje izvorne kode ali dostop do razvojne infrastrukture. Zaradi tega lahko prevzem računa predstavlja bistveno večje tveganje kot napad na posamezen strežnik. Če napadalec pridobi dostop do računa legitimnega vzdrževalca, so lahko spremembe v programski opremi videti popolnoma zaupanja vredne. Namesto neposrednega napada na uporabnike napadalec izkoristi obstoječi odnos zaupanja med vzdrževalcem, registrom paketov in končnimi uporabniki.

Eden najpogostejših načinov prevzema računov je kraja prijavnih podatkov prek lažnega predstavljanja (angl. phishing). Napadalci ustvarijo lažne prijavne strani ali komunikacijo, ki posnema legitimne storitve, kot so GitHub, npm ali drugi razvojni sistemi. Cilj je pridobiti uporabniško ime, geslo ali dodatne podatke za preverjanje identitete. Ker številne organizacije uporabljajo večfaktorsko avtentikacijo (angl. Multi-Factor Authentication, MFA), napadalci vse pogosteje uporabljajo naprednejše tehnike, ki poskušajo pridobiti tudi enkratne kode ali uporabnika preusmeriti skozi lažen prijavni proces. S tem lahko zaobidejo tradicionalno zaščito in pridobijo dostop do računa, ki ima dovoljenja za objavo programske opreme.

Poleg gesel predstavljajo pomembno tveganje tudi dostopni žetoni (angl. access tokens). Ti omogočajo avtomatiziran dostop do storitev brez ponovne prijave uporabnika. Če napadalec pridobi veljaven žeton, lahko pogosto izvaja dejanja v imenu uporabnika, dokler žeton ni preklican. To je posebej nevarno v razvojnih okoljih, kjer so računi vzdrževalcev pogosto povezani z več storitvami, kot so repozitoriji, registri paketov, sistemi CI/CD in oblačne platforme. Ukraden žeton lahko tako predstavlja vstopno točko do več različnih delov dobavne verige.

Odprtokodni vzdrževalci so zaradi svoje vloge pogosto zanimive tarče. Napadalci lahko uporabijo tehnike socialnega inženiringa (angl. social engineering), pri katerih poskušajo uporabnika prepričati, da izvede določeno dejanje, na primer odpre povezavo, namesti orodje ali potrdi zahtevo za dostop. Problem je še posebej izrazit pri priljubljenih odprtokodnih projektih, kjer lahko en sam vzdrževalec nadzoruje paket, ki ga uporablja veliko število aplikacij. Napadalcu zato ni treba iskati ranljivosti v vseh ciljnih sistemih; dovolj je prevzeti račun ene osebe, ki ima ustrezne pravice.

Incident event-stream, podrobneje predstavljen v poglavju 3.1, kaže tudi na nevarnost socialnega prevzema vzdrževanja projekta. Napadalec je pridobil zaupanje prvotnega vzdrževalca in nato prek legitimnega postopka objavil zlonamerno odvisnost. Nato je v paket vključil novo odvisnost flatmap-stream, ki je vsebovala zlonamerno kodo. Ker je bila sprememba objavljena prek legitimnega računa in zaupanja vrednega projekta, se je razširila po običajnem postopku posodabljanja.

Podoben vzorec se je pojavil tudi pri napadih na pakete ua-parser-js (2021), chalk, debug in eslint-config-prettier (2025), kjer so napadalci po prevzemu računov vzdrževalcev objavili zlonamerne različice priljubljenih knjižnic. Takšni primeri kažejo, da lahko prevzet račun razvijalca predstavlja učinkovitejšo vstopno točko kot neposreden napad na infrastrukturo [18], [19].

Posebej odmevna je bila kampanja Shai-Hulud leta 2025. Za razliko od posameznih zlonamerno spremenjenih paketov je šlo za kampanjo, pri kateri je kompromitirana koda v paketih avtomatizirano iskala dodatne skrivnosti, dostopne žetone in druge poverilnice (angl. credentials) ter jih uporabljala za nadaljnje širjenje. Po prevzemu računov vzdrževalcev so bile objavljene zlonamerne različice paketov, ki so omogočile nadaljnje širjenje napada na povezane pakete in razvojna okolja. Takšen način delovanja kaže značilnosti črva (angl. worm), kjer posamezna uspešna kompromitacija ustvari pogoje za naslednje faze napada [18].

Podoben primer predstavlja napad na projekte, ki uporabljajo GitHub Actions za objavo paketov na PyPI. Napadalci so po pridobitvi dostopa do izvorne kode spremenili delovne tokove GitHub Actions in dodali korake,

namenjene kraji žetonov za objavo PyPI (angl. PyPI API tokens). Ob naslednji izvedbi delovnega toka so se žetoni prenesli napadalcem, ki bi jih lahko uporabili za objavo zlonamernih različic paketov. Čeprav PyPI kot platforma ni bila ogrožena, je incident pokazal, kako lahko kompromitiran razvojni proces ogrozi celotno dobavno verigo [20].

### 3.3. Zloraba okolij CI/CD

Sistemi CI/CD predstavljajo osrednji del sodobnega razvoja programske opreme. Omogočajo avtomatizirano gradnjo, testiranje in objavo aplikacij, s čimer razvojne ekipe hitreje in z manj ročnega dela dostavljajo spremembe v produkcijska okolja. Ravno zaradi svoje vloge so okolja CI/CD privlačna tarča za napadalce. Med izvajanjem razvojnih procesov imajo pogosto dostop do izvorne kode, skrivnosti (angl. secrets), dostopnih žetonov (angl. access tokens), podpisnih ključev in poverilnic (angl. credentials) za dostop do oblačne infrastrukture. Zloraba okolja CI/CD zato lahko napadalcu omogoči dostop do večjega števila sistemov, ne da bi moral neposredno napasti produkcijsko okolje [21], [22].

Eden najpogostejših ciljev napadalcev pri napadih na okolja CI/CD je odtekanje skrivnosti (angl. secret exfiltration). Razvojni procesi pogosto potrebujejo občutljive podatke za dostop do zunanjih storitev, na primer API ključev, poverilnic za oblačne ponudnike ali žetonov za objavo paketov. Če napadalec pridobi dostop do teh podatkov, lahko pogosto nadaljuje napad brez ponovnega izkoriščanja ranljivosti. Ukradene poverilnice lahko uporabi za dostop do repozitorijev, spreminjanje programske kode, objavo zlonamernih paketov ali dostop do produkcijske infrastrukture [23], [24].

Poseben problem predstavlja dejstvo, da so CI/CD sistemi pogosto zasnovani tako, da imajo obsežnejše pravice dostopa kot posamezni razvijalci. Njihov namen je avtomatizacija, zato morajo imeti dovoljenja za izvajanje operacij, ki jih običajni uporabniki ne bi smeli izvajati. Če napadalec prevzame tak proces, lahko zlorabi že obstoječe pravice.

Napadalci lahko okolja CI/CD zlorabijo na več načinov. Eden izmed pogostih pristopov je sprememba delovnih tokov (angl. workflows) oziroma konfiguracijskih datotek, ki določajo, katere ukaze sistem izvede med gradnjo in objavo programske opreme. Če napadalec pridobi dostop do repozitorija ali razvojnega računa, lahko v CI/CD proces doda dodatne ukaze, ki se izvajajo skupaj z legitimnimi opravili. Ti ukazi lahko na primer zbirajo okoljske spremenljivke, izvozijo skrivnosti ali pošiljajo ukradene podatke napadalcu. Ker se vse izvaja v okviru legitimnega razvojnega procesa, je takšne spremembe težje opaziti [21], [23]. Enaka načela veljajo za različne platforme, kot so GitHub Actions, GitLab CI in Azure DevOps. Razlike med posameznimi sistemi obstajajo, vendar je skupna točka enaka: CI/CD predstavlja zaupanja vredno okolje, ki ima dostop do številnih drugih sistemov.

Največja nevarnost ogroženega okolja CI/CD je možnost prehoda iz razvojnega procesa v produkcijsko okolje. V tradicionalnih napadih napadalec običajno išče ranljivosti neposredno v produkcijskih sistemih. Pri napadih na dobavno verigo pa lahko izkoristi razvojni proces kot vstopno točko. Če CI/CD sistem uporablja poverilnice za dostop do produkcije, lahko napadalec po zlorabi CI/CD cevovoda pridobi možnost nameščanja spremenjene programske opreme, spreminjanja infrastrukture ali dostopa do oblačnih storitev. Razvojni proces tako postane most med izvorno kodo in produkcijo [21], [22]. Zato zaščita okolij CI/CD ni več samo vprašanje varnosti razvojnih orodij, temveč predstavlja pomemben del celotne varnostne strategije organizacije.

Eden izmed novejših primerov predstavlja val napadov, povezanih z ekosistemom Trivy leta 2026, kjer so napadalci spremenili komponente, ki se uporabljajo v CI/CD procesih za varnostno preverjanje vsebnikov. Napadalci so izkoristili legitimno orodje in vanj vključili kodo za krajo poverilnic. Zlonamerna različica je poskušala pridobiti predvsem poverilnice za oblačna okolja, kot so AWS ključi in druge skrivnosti, ki so bile dostopne v CI/CD izvajalnih okoljih. Incident je pokazal, da lahko tudi varnostna orodja sama postanejo del napadalne površine, če imajo širok dostop do razvojnih procesov [25].

Podoben primer predstavlja napad na orodje Checkmarx KICS leta 2026. Napadalci jim je uspelo objaviti zlonamerno spremenjene različice artefaktov, povezanih z orodjem, vključno z Docker slikami in razširitvami za

razvojna okolja. Zlonamerne različice so bile namenjene kraji podatkov iz razvojnih okolij, vključno s skrivnostmi in drugimi občutljivimi informacijami, ki jih orodja za varnostno analizo pogosto obdelujejo. Primer kaže, da lahko napad na razvojno orodje omogoči dostop do informacij, ki so običajno zaščitene v razvojnem procesu [25].

Dodaten primer predstavlja vzorec ranljivosti Cordyceps iz leta 2026, pri katerem so raziskovalci pokazali, da lahko napačne konfiguracije delovnih tokov CI/CD omogočijo dostop do velikega števila repozitorijev GitHub. Analiza je pokazala, da lahko takšni avtomatizirani razvojni procesi razkrijejo dostop do izvorne kode, skrivnosti in drugih privilegiranih virov. Primer ne predstavlja potrjene množične napadalne kampanje, temveč kaže, kako lahko pomanjkljivo zaščiteni procesi CI/CD ustvarijo pomembno tveganje za programsko dobavno verigo [26].

### 3.4. Identitetni sistemi in OIDC

V sodobnih razvojnih okoljih identiteta postaja ena od najpomembnejših varnostnih točk. Medtem ko so se tradicionalni napadi pogosto osredotočali na krajo gesel ali skrivnosti (angl. secrets), sodobna oblačna okolja vse pogosteje temeljijo na preverjanju identitete uporabnikov, aplikacij in avtomatiziranih procesov. To je še posebej pomembno pri okoljih CI/CD, kjer razvojni sistemi potrebujejo dostop do drugih storitev, na primer oblačne infrastrukture, registrov paketov ali produkcijskih okolij. Namesto shranjevanja dolgoročnih poverilnic se danes pogosto uporablja federirana identiteta (angl. federated identity), pri kateri ena storitev preveri identiteto druge in ji dodeli začasen dostop.

Federirana identiteta omogoča, da sistem ne uporablja neposredno shranjenih uporabniških imen, gesel ali trajnih ključev, temveč zaupa drugi identitetni storitvi. V okoljih CI/CD je pogost primer povezava med platformo, kot je GitHub Actions, in oblačnim ponudnikom, kot so AWS, Azure ali Google Cloud. Namesto da bi imel razvojni sistem shranjen trajni dostopni ključ, lahko pridobi kratkoživi žeton (angl. short-lived token), ki velja samo omejen čas in samo za določeno dejanje. OIDC je eden izmed standardov, ki omogoča takšno izmenjavo identitete med različnimi sistemi [11], [27].

Tipičen proces poteka tako, da sistem CI/CD najprej pridobi OIDC žeton, ki vsebuje informacije o identiteti izvajalnega okolja. Oblačni ponudnik nato preveri, ali tej identiteti zaupa in ali izpolnjuje določene pogoje. Če je preverjanje uspešno, izda začasne poverilnice z omejenimi pravicami. Ta pristop zmanjšuje potrebo po dolgoročnih skrivnostih, vendar hkrati uvaja novo varnostno tveganje. Če je konfiguracija zaupanja napačna ali če napadalec pridobi nadzor nad zaupanja vrednim procesom, lahko pridobi legitimni dostop do drugih sistemov.

Napadi na identitetne sisteme se pogosto ne osredotočajo na samo tehnologijo OIDC, temveč na napačne konfiguracije in preširoko nastavljene pravice dostopa. Primer je lahko okolje CI/CD, ki ima dovoljenje za dostop do oblačnega računa brez dovolj strogih omejitev glede izvora zahtevka. Napadalec lahko v takem primeru poskuša zlorabiti razvojni proces, ustvariti veljaven zahtevek iz zaupanja vrednega okolja ali izkoristiti preširoko konfigurirano identitetno povezavo. Ker sistem vidi veljavno identiteto, lahko napad deluje kot legitimna aktivnost [11].

Pomemben koncept pri tem je identiteta delovne obremenitve (angl. Workload Identity). Gre za mehanizem, pri katerem aplikacijam in avtomatiziranim procesom dodelimo lastne identitete, podobno kot jih imajo uporabniki. Prednost je manjše število skrivnosti, ki jih je treba shranjevati in varovati, vendar mora biti upravljanje dovoljenj zelo natančno. Preširoka dovoljenja lahko povzročijo, da ogrožen proces dobi dostop do veliko več virov, kot bi jih potreboval. Zaradi tega identitetni sistemi postajajo nova osrednja točka zaupanja. Napadalcu ni več nujno treba ukrasti gesla ali neposredno napasti produkcijskega strežnika. Dovolj je, da zlorabi legitimno identiteto, ki ji sistem že zaupa.

Eden izmed pomembnih primerov je kampanja, povezana z ekosistemom TanStack leta 2026, kjer so napadalci zlorabili razvojni proces in prek njega objavili zlonamerne različice več npm paketov, med njimi tudi priljubljene pakete iz ekosistema TanStack. Napad je izkoristil povezavo med okoljem CI/CD, npm registrom in identitetnimi mehanizmi za avtomatizirano objavo paketov. Ker je proces objave uporabljal federacijo OIDC, je incident pokazal, da lahko ogrožen izvajalec CI/CD pridobi veljavno razvojno identiteto in jo uporabi za legitimno objavo paketov. Posledica ni bila samo objava zlonamernih različic posameznih paketov, temveč tveganje za vse

uporabnike, ki so zlonamerne različice prenesli prek običajnega postopka posodabljanja. Po incidentu so vzdrževalci omejili dovoljenja, izboljšali nadzor nad objavo paketov in utrdili konfiguracijo CI/CD. Primer kaže, da kratkoživi žetoni sami po sebi ne rešijo problema, če lahko ogrožen proces dostopa do identitete z dovoljenji za objavo [27], [28].

Podoben primer predstavlja kampanja Megalodon, kjer so napadalci zlorabili GitHub okolje in avtomatizirane CI/CD procese za krajo poverilnic ter nadaljnje širjenje napada. Ključna značilnost napada je bila, da ukradeni podatki niso predstavljali samo trenutnega dostopa do posameznega sistema, temveč so napadalcem omogočili pridobivanje novih identitet in dostopov. Na primer, ukradeni žetoni, skrivnosti in dostopne poverilnice iz razvojnih okolij so lahko omogočili dostop do dodatnih repozitorijev, oblačnih storitev ali drugih CI/CD okolij. S tem je ogrožen razvojni proces postal izvor novih zaupanja vrednih dostopov, ki jih je bilo mogoče uporabiti za nadaljevanje napada po dobavni verigi [29].

### 3.5. Zloraba kontejnerskih slik (angl. container images)

Kontejnerji (angl. containers) so danes eden izmed temeljev sodobnega razvoja in uvajanja programske opreme. Razvijalcem omogočajo, da aplikacijo skupaj z vsemi potrebnimi odvisnostmi zapakirajo v prenosljivo enoto, ki jo je mogoče enako izvajati v različnih okoljih. Osnovni gradnik takšnega okolja je kontejnerska slika (angl. container image), ki vsebuje aplikacijsko kodo, knjižnice, sistemske komponente in druge potrebne datoteke.

Podobno kot pri programskih odvisnostih velja, da organizacije pogosto ne ustvarijo vseh komponent same, temveč uporabljajo že pripravljene slike iz javnih registrov, kot je Docker Hub. S tem nastane nov odnos zaupanja: organizacija zaupa izvoru slike, njenim vzdrževalcem in vsem komponentam, ki so vključene v njeno vsebino. Če je ena izmed teh komponent ogrožena, lahko napadalec doseže vse uporabnike, ki jo vključijo v svoje razvojne ali produkcijske procese [30].

Napadalci lahko kontejnerske slike zlorabijo na več načinov. Eden izmed najpogostejših pristopov je objava zlonamernih slik v javnih registrih. Napadalec lahko ustvari sliko z legitimno vidnim imenom, vanjo vključi zlonamerno kodo in poskuša doseči, da jo razvijalci uporabijo kot osnovo za svoje aplikacije. Podoben problem predstavljajo kompromitirane osnovne slike (angl. base images). Pri gradnji kontejnerjev razvijalci pogosto uporabljajo obstoječe slike kot izhodišče, na primer operacijski sistem ali že pripravljeno razvojno okolje. Če vsebuje takšna osnovna slika zlonamerno kodo ali ranljive komponente, se težava prenese v vse nadaljnje slike, ki so zgrajene na njej [30], [31].

Dodatno tveganje predstavlja uporaba spremenljivih oznak (angl. mutable tags), kot je na primer oznaka latest. Če se vsebina slike za isto oznako spremeni, lahko razvojna ekipa ali sistem CI/CD nehote prenese drugačno različico, kot je bila prvotno preverjena. Zaradi tega sodobne prakse priporočajo uporabo preverljivih različic, podpisovanje artefaktov in preverjanje izvora komponent [30].

Kontejnerske slike so pogosto tesno povezane s CI/CD procesi. Tipičen razvojni tok vključuje samodejno gradnjo slike, varnostno preverjanje, objavo v register in kasnejšo namestitev v produkcijsko okolje. Če napadalec zlorabi del tega procesa, lahko vpliva na končno sliko, ki jo organizacija uporabi v produkciji. To pomeni, da napad ni nujno usmerjen neposredno v delujočo aplikacijo, temveč v proces, ki aplikacijo izdela in dostavi. Zaradi tega predstavljajo registri kontejnerskih slik in procesi njihove gradnje pomemben del programske dobavne verige. Ogrožena slika lahko omogoči izvajanje zlonamerne kode, krajo skrivnosti (angl. secrets) ali pridobivanje dostopa do notranjih sistemov [32], [33].

Eden izmed pogostih primerov so kampanje, kjer napadalci objavijo zlonamerne slike v javnih registrih, kot je Docker Hub. Raziskave podjetja Sysdig so pokazale številne primere slik, ki so vsebovale skrite rudarje kriptovalut, mehanizme za nepooblaščen dostop (angl. backdoors) ali drugo neželeno funkcionalnost. Ker so slike pogosto delovale legitimno in so bile javno dostopne, so jih lahko uporabniki vključili v svoja okolja brez zavedanja o tveganju [32].

Podoben trend so pokazale novejše kampanje, povezane s krajo poverilnic v okoljih CI/CD. Napadi na orodja, ki se uporabljajo pri gradnji in preverjanju kontejnerskih slik, so pokazali, da lahko ogrožena komponenta razvojnega procesa omogoči dostop do skrivnosti in drugih občutljivih podatkov. Primeri, povezani s Trivy in KICS, kažejo, da tudi varnostna orodja in infrastruktura za preverjanje lahko postanejo del napadalne površine [33], [34].

# 4 Večstopenjske napadalne verige

V prejšnjem poglavju smo predstavili najpogostejše napadalne tehnike, ki se uporabljajo pri napadih na programske dobavne verige. Sodobni napadi na programske dobavne verige pogosto niso izvedeni kot en sam korak, temveč kot zaporedje povezanih dejanj. Posamezna faza napadalcu omogoči pridobitev novih pravic ali dostopa, ki ga uporabi za naslednjo stopnjo napada. Primer takšnega vzorca predstavlja aktivnost, povezana s TeamPCP in kampanjo Shai-Hulud, kjer so različni napadalni valovi izkoristili povezave med razvojnimi okolji, paketnimi ekosistemi in avtomatiziranimi procesi objave. Namesto ločenih incidentov jih je zato smiselno obravnavati kot večstopenjsko verigo, v kateri zloraba ene komponente omogoči nadaljnje širjenje prek novih identitet, žetonov in distribucijskih kanalov. Tako nastane večstopenjska napadalna veriga, ki napadalcem omogoča postopno širjenje dostopa skozi razvojna, testna in produkcijska okolja.

Posebnost takšnih napadov je, da napadalci pogosto ne poskušajo neposredno zaobiti vseh varnostnih mehanizmov. Namesto tega izkoristijo obstoječe odnose zaupanja med različnimi deli razvojnega procesa. Razvojni račun zaupa repozitoriju, proces CI/CD zaupa izvornemu okolju, register paketov zaupa postopku objave, uporabniki pa zaupajo programski opremi, ki prihaja iz legitimnega vira. Če napadalec prevzame eno izmed teh povezav, lahko pridobljeni dostop uporabi za nadaljnje korake napada.

## 4.1. Tipični potek sodobnega napada

Sodobni napad na programsko dobavno verigo se praviloma začne z ogrožanjem zaupanja vredne komponente, računa ali razvojnega procesa. Začetni dostop napadalcu omogoči prehod v okolje CI/CD, kjer lahko pridobi skrivnosti, žetone za objavo, oblačne poverilnice ali druge identitete z dostopnimi pravicami.

Pridobljene identitete nato uporabi za dostop do dodatnih repozitorijev, registrov, oblačnih storitev in produkcijskih okolij oziroma za objavo zlonamernih različic programske opreme. Posamezne faze napada niso ločene, temveč tvorijo verigo, v kateri vsak uspešen korak omogoči nadaljnje širjenje vpliva napadalca.

## 4.2. Kaskadni učinki med organizacijami

Posebnost napadov na programske dobavne verige je, da posledice pogosto presegajo eno organizacijo. Medtem ko klasični kibernetski napadi običajno prizadenejo posamezno tarčo, lahko ogrožanje zaupanja vrednega ponudnika programske opreme vpliva na več sto ali celo več tisoč organizacij.

Če napadalec zlorabi priljubljen odprtokodni paket, razvojno orodje ali ponudnika programske opreme, lahko zlonamerna koda doseže vse organizacije, ki uporabljajo takšno rešitev. Vsaka od teh organizacij lahko nato nevede postane nova vstopna točka za nadaljnje napade, kar povzroči kaskadni učinek po celotni dobavni verigi.

Takšen način širjenja je bil značilen za več odmevnih incidentov, kot so event-stream, ua-parser-js, SolarWinds, XZ Utils in novejše kampanje, usmerjene v npm ter GitHub ekosistem. Čeprav so se napadi med seboj razlikovali po začetnem vektorju, je bila njihova skupna značilnost zloraba zaupanja v programsko dobavno verigo in posledično širjenje vpliva daleč prek prvotne tarče [21], [22], [35], [36].

Napadi na programske dobavne verige zato ne predstavljajo več zgolj tehničnega problema posamezne organizacije, temveč sistemsko tveganje za celoten programski ekosistem. Vse večja povezanost razvojnih orodij, odprtokodnih projektov, oblačnih storitev in avtomatiziranih razvojnih procesov pomeni, da lahko zloraba ene same zaupanja vredne komponente vpliva na veliko število medsebojno povezanih organizacij.

# 5 Vpliv umetne inteligence na varnost dobavnih verig

Generativna umetna inteligenca je v zadnjih letih postala pomemben del razvoja programske opreme. Orodja, zasnovana na velikih jezikovnih modelih (angl. Large Language Models, LLM), razvijalcem pomagajo pri dopolnjevanju kode, ustvarjanju funkcij, pisanju testov, razlagi obstoječih rešitev, pripravi dokumentacije in avtomatizaciji ponavljajočih se opravil. Z razvojem sistemov z večjo stopnjo avtonomije se njihova uporaba širi tudi na bolj kompleksne naloge: sodobni razvojni agenti lahko samostojno pregledujejo repozitorije, spreminjajo več datotek, nameščajo odvisnosti, zaganjajo ukaze in pripravljajo zahtevke za vključitev sprememb (angl. pull requests).

Takšna orodja spreminjajo programsko dobavno verigo predvsem zato, ker povečujejo hitrost in obseg nastajanja programske kode. Več sprememb je mogoče pripraviti v krajšem času, vendar se posledično poveča količina kode, ki jo morajo preveriti razvijalci, varnostni strokovnjaki in avtomatizirana orodja. Umetna inteligenca (v nadaljevanju UI) zato ni samostojen vzrok napadov na dobavne verige, temveč pospeševalec že obstoječih razvojnih procesov, napak in razmerij zaupanja. Če so postopki pregleda, upravljanja odvisnosti ali omejevanja privilegijev šibki, lahko umetna inteligenca poveča hitrost, s katero se te pomanjkljivosti odrazijo v končnih programskih artefaktih.

Uporaba umetne inteligence ima na varnost dvojni vpliv. Na eni strani lahko modeli pomagajo pri iskanju napak, generiranju testov, analizi kode in razlagi ranljivosti. Na drugi strani lahko ustvarijo funkcionalno delujočo, vendar varnostno neustrezno kodo, predlagajo neobstoječe odvisnosti ali izvedejo neželena dejanja, kadar so povezani z razvojnimi orodji in privilegiranimi računi. Ključno vprašanje zato ni, ali naj se UI uporablja, temveč kako jo vključiti v razvojni proces, ne da bi pri tem zmanjšali preverljivost, sledljivost in človeški nadzor. NIST pri upravljanju tveganj UI poudarja potrebo po merjenju tveganj, dokumentiranju omejitev, neodvisnem preverjanju rezultatov in jasno določenih odgovornostih skozi celoten življenjski cikel sistema [37], [38].

## 5.1. Umetna inteligenca kot pospeševalec razvoja

Najbolj razširjena uporaba UI v razvoju programske opreme je pomoč pri pisanju kode. Razvojno orodje na podlagi komentarja, opisa funkcionalnosti ali obstoječega konteksta predlaga posamezne vrstice, celotne funkcije ali večje programske komponente. Eksperimentalne raziskave ponudnikov takšnih orodij so pokazale, da lahko razvijalci določene naloge dokončajo hitreje, pri čemer je treba rezultate razlagati v okviru konkretnih eksperimentov in ob upoštevanju dejstva, da gre pogosto za raziskave ponudnikov samih [39]. Ne glede na dejanski obseg vpliva je jasno, da umetna inteligenca zmanjšuje čas, potreben za izdelavo začetne različice rešitve, in s tem povečuje število sprememb, ki jih je mogoče vključiti v razvojni tok.

UI se uporablja tudi za ustvarjanje testov enot in integracijskih testov. Model lahko iz podane funkcije pripravi osnovne testne primere, predlaga mejne vrednosti ali dopolni manjkajočo testno dokumentacijo. NIST je leta 2025 začel pilotno vrednotenje zmožnosti UI za izdelavo testov posameznih enot (angl. unit tests), kar kaže, da se uporaba modelov širi iz neposrednega pisanja produkcijske kode tudi v postopke zagotavljanja kakovosti [40]. Takšna avtomatizacija je koristna predvsem kot začetna pomoč, vendar kakovost testov ostaja odvisna od pravilnosti specifikacij, izbranih testnih primerov in neodvisnega preverjanja, ali testi dejansko zaznajo pomembne napake.

Pomembno področje uporabe so pregledi kode. Modeli lahko povzamejo spremembe, opozorijo na sumljive vzorce, predlagajo izboljšave ali razvijalcu razložijo posledice določene implementacije. S tem lahko zmanjšajo kognitivno obremenitev pri pregledovanju obsežnih zahtevkov za vključitev sprememb (angl. pull requests). Toda pregled kode s podporo UI ne sme postati nadomestilo za tehnični in varnostni pregled, saj model pogosto ocenjuje verjetnost in podobnost z naučenimi vzorci, ne pa dejanske varnostne lastnosti sistema. Njegov izhod je zato priporočilo, ki ga je treba preveriti z analiznimi orodji, testi in strokovnim pregledom.

Novejša stopnja razvoja so agentni sistemi, ki niso omejeni na generiranje besedila ali kode. Tak agent lahko deluje z dostopom do repozitorija, terminala, registra paketov, sistema za sledenje nalogam in okolja CI/CD. Na podlagi enega navodila lahko analizira projekt, spremeni kodo, namesti novo odvisnost, zažene teste in pripravi

spremembo za objavo. S tem se meja med svetovalnim orodjem in aktivnim udeležencem programske dobavne verige zabriše. Agent postaja nov izvajalec z lastno identiteto in dovoljenji oziroma posrednik, katerega dovoljenja, dejanja in dostop do podatkov je treba obravnavati podobno kot dovoljenja razvijalcev in avtomatiziranih CI/CD procesov.

Umetna inteligenca lahko varnost dobavne verige tudi neposredno izboljšuje. Uporablja se lahko za razvrščanje opozoril, analizo sprememb v odvisnostih, pomoč pri razumevanju ranljivosti in prepoznavanje nepričakovanih vzorcev v dnevniških zapisih. Pri tem je najbolj učinkovit v vlogi dodatnega analiznega sloja, ki pomaga prednostno razporediti delo in opozori na odstopanja. Vendar mora biti odločanje o sprejemu odvisnosti, objavi artefakta ali dostopu do produkcije še vedno povezano z določljivimi pravili, preverljivimi dokazi in neodvisnimi varnostnimi kontrolami.

Glavna posledica UI-podprtega razvoja je torej povečanje hitrosti. Organizacija lahko hitreje ustvarja kodo, teste in konfiguracije, vendar mora enako hitro razširiti tudi postopke preverjanja. Če se produktivnost razvijalcev poveča, medtem ko zmogljivosti pregledovanja, testiranja in spremljanja ostanejo nespremenjene, nastane nesorazmerje: v dobavno verigo vstopa več sprememb, kot jih je mogoče kakovostno oceniti. V takem okolju lahko tudi majhen delež napačnih ali nevarnih predlogov povzroči pomembno povečanje skupnega tveganja.

### 5.2. Nova tveganja zaradi umetne inteligence

Prvo pomembno tveganje je sprejemanje nepreverjene kode. Koda, ki jo ustvari UI, je lahko sintaktično pravilna, funkcionalna in na prvi pogled smiselna, vendar vsebuje ranljivosti, napačne predpostavke ali zastarele varnostne vzorce. Raziskava Pearcea in sodelavcev je pri varnostno občutljivih scenarijih pokazala, da je pomemben delež analiziranih predlogov orodja GitHub Copilot vseboval ranljivosti [41]. Kasnejše ponovitvene raziskave so sicer ugotovile izboljšanje rezultatov, vendar so še vedno zaznale varnostno neustrezne predloge [42]. Takšne ugotovitve ne pomenijo, da je vsa z UI ustvarjena koda nevarna, temveč da samo dejstvo, da je kodo ustvaril model, ne predstavlja varnostnega zagotovila.

Tveganje se poveča, kadar razvijalec uporablja tehnologijo, ki je ne pozna dovolj dobro. Model lahko pripravi rešitev, ki deluje v običajnem primeru, razvijalec pa zaradi pomanjkanja izkušenj ne prepozna neustreznega preverjanja vhodov, šibke avtentikacije, nevarnega upravljanja skrivnosti ali napačne uporabe kriptografskih funkcij. Prepričljivo oblikovan odgovor lahko ustvari vtis pravilnosti. Zato mora za kodo, ki jo ustvari UI, veljati najmanj enaka raven pregleda kot za kodo zunanjega sodelavca ali novo odvisnost neznanega izvora.

Drugo tveganje predstavljajo izmišljene programske odvisnosti (angl. package hallucinations). Model lahko pri generiranju kode predlaga paket, ki v registru npm, PyPI ali drugem ekosistemu sploh ne obstaja. Napadalec lahko takšno predvidljivo ime registrira in vanj vstavi zlonamerno kodo. Če razvijalec slepo sledi predlogu modela, namesti napadalčev paket. Ta tehnika se pogosto imenuje slopsquatting in predstavlja posebno različico zastrupljanja paketov, pri kateri napadalec izkorišča ponavljajoče se napake generativnih modelov.

Spracklen in sodelavci so v obsežni raziskavi analizirali 576.000 vzorcev kode, ustvarjenih s šestnajstimi modeli, in ugotovili, da so halucinacije paketov sistemski pojav, ki se razlikuje glede na model, programski jezik in način postavitve vprašanja. Raziskava je pri komercialnih modelih izmerila najmanj 5,2 %, pri odprtokodnih modelih pa 21,7 % povprečni delež haluciniranih paketov [43]. Posamezna napačna navedba še ne pomeni uspešnega napada, vendar ponavljajoča se imena ustvarjajo priložnosti za registracijo zlonamernih paketov. Obramba zahteva preverjanje obstoja, avtorja, zgodovine in izvora vsake nove odvisnosti ter uporabo dovoljenih seznamov in zaklenjenih različic.

Tretje tveganje je vbrizgavanje navodil (angl. prompt injection) pri razvojnih agentih. Klasičen pomočnik prejema navodila neposredno od uporabnika, agent pa pogosto bere še izvorno kodo, dokumentacijo, težave v sistemu za sledenje, spletne strani ali vsebino zunanjih paketov. Napadalec lahko v enega od teh virov vstavi skrita ali zavajajoča navodila, ki poskušajo preusmeriti vedenje agenta. Pri posrednem vbrizgavanju agent obravnava napadalčevo vsebino kot navodilo, čeprav je bila prvotno le podatek za analizo. Posledice takšnega napada so bistveno resnejše, kadar ima agent dostop do orodij. Namesto napačnega besedilnega odgovora lahko spremeni

datoteko, zažene ukaz, prenese paket, pošlje vsebino zunanji storitvi ali uporabi poverilnice iz razvojnega okolja. OWASP med ključnimi tveganji za sisteme z umetno inteligenco, ki delujejo avtonomno, izpostavlja neposredno in posredno vbrizgavanje navodil, zlorabo orodij, stopnjevanje privilegijev ter odtekanje podatkov [44]. S tem razvojni agent postane nova povezava v verigi zaupanja: zaupa vhodnim podatkom, uporabnik zaupa njegovim odločitvam, infrastruktura pa zaupa identiteti in dovoljenjem, s katerimi agent deluje.

Četrto tveganje predstavlja zmanjšanje človeškega nadzora. Večja količina samodejno ustvarjene kode lahko privede do površinskih pregledov, zlasti kadar so spremembe obsežne in navidezno rutinske. Razvijalec lahko potrdi rezultat, ker testi uspešno prestanejo izvajanje, ne da bi razumel vse ustvarjene odvisnosti, konfiguracije ali stranske učinke. Problem ni le v kakovosti posamezne vrstice kode, temveč v izgubi sledljivosti: težje je ugotoviti, zakaj je bila določena arhitekturna odločitev sprejeta, na podlagi katerih podatkov oziroma vzorcev je model pripravil predlog in kdo je dejansko preveril njene posledice.

Orodja, podprta z UI lahko povzročijo tudi nenamerno razkritje podatkov. Razvijalci lahko v poziv vključijo izvorno kodo, konfiguracije, osebne podatke, notranjo dokumentacijo ali skrivnosti. Pri agentnih sistemih je tveganje še večje, saj lahko orodje samodejno bere datoteke in kontekst repozitorija. Organizacija mora zato določiti, kateri podatki se lahko posredujejo modelu, kako dolgo se hranijo, ali se uporabljajo za nadaljnje učenje in katere zunanje storitve jih obdelujejo. Profil NIST za generativno umetno inteligenco obravnava upravljanje podatkov, zasebnost, varnost, nadzor tretjih ponudnikov in neodvisno vrednotenje kot medsebojno povezane elemente upravljanja tveganj [37].

Nazadnje lahko generativna UI pospeši tudi delo napadalcev. Uporablja se lahko za prilagajanje phishing sporočil, analizo izvorne kode, pomoč pri ustvarjanju zlonamernih paketov ali njihovih različic ali avtomatizacijo iskanja napačnih konfiguracij. Vendar osnovni napadalni mehanizmi večinoma ostajajo enaki: kraja identitet, zloraba zaupanja, zloraba odvisnosti in izkoriščanje preširokih privilegijev. UI zmanjšuje stroške in čas izvedbe, ne odpravlja pa potrebe po začetni vstopni točki. Zaradi tega morajo obrambni ukrepi ostati usmerjeni v preverjanje izvora, omejevanje privilegijev, zaščito razvojnih identitet, nadzor sprememb in zaznavanje nepričakovanega vedenja.

# 6 Obrambni pristopi

Ker napadi na programske dobavne verige izkoriščajo odnose zaupanja med komponentami, identitetami in razvojnimi procesi, zaščita ne more temeljiti na enem samem ukrepu. Organizacije morajo poznati uporabljene komponente, preverjati njihov izvor in celovitost ter omejevati dostop ljudi in avtomatiziranih procesov.

Obrambne pristope lahko razdelimo na tehnične in organizacijske ukrepe. Tehnični ukrepi vključujejo izdelavo SBOM, upravljanje odvisnosti, preverjanje porekla artefaktov, zaščito okolij CI/CD, upravljanje identitet in varnost kontejnerskih slik. Organizacijski ukrepi pa zajemajo upravljanje privilegijev, nadzor dobaviteljev, vključevanje varnostnih pregledov v razvojni proces, stalno spremljanje tveganj in pripravo odziva na incidente. Ker se učinkovitost posameznih ukrepov razlikuje glede na okolje in razpoložljive vire, je pomembno, da organizacije ukrepe uvajajo postopoma in glede na njihovo tveganje ter kritičnost.

## 6.1. Tehnični ukrepi

Tehnična zaščita programske dobavne verige temelji na povečanju preglednosti in zaupanja v procesu razvoja programske opreme. Sodobne aplikacije niso več sestavljene samo iz kode, ki jo napiše organizacija sama, temveč vključujejo številne zunanje knjižnice, pakete, kontejnerske slike, razvojna orodja in avtomatizirane procese. Zaradi tega mora zaščita zajemati celoten življenjski cikel programske opreme: od izbire komponent, razvoja in gradnje do dostave v produkcijsko okolje.

Glavni cilj tehničnih ukrepov je odgovoriti na tri vprašanja: katere komponente uporabljamo, od kod prihajajo in ali lahko preverimo, da med procesom razvoja niso bile spremenjene. V nadaljevanju so predstavljeni najpomembnejši pristopi, ki se uporabljajo za povečanje zaupanja v programsko dobavno verigo.

### *6.1.1. SBOM, upravljanje odvisnosti in varnost paketnih ekosistemov*

Eden izmed osnovnih pristopov je izdelava kosovnice programske opreme (Software Bill of Materials). SBOM je torej popis komponent, ki sestavljajo določeno različico programske opreme. Običajno nastane kot del procesa gradnje programske opreme in vsebuje informacije o uporabljenih knjižnicah, paketih, njihovih različicah ter drugih komponentah. Organizacije ga lahko hranijo skupaj s programskim artefaktom ali ga vključijo v svoje razvojne in varnostne procese. SBOM omogoča pregled nad programsko dobavno verigo določene aplikacije: katere komponente so vključene, katere različice se uporabljajo in od kod prihajajo [45]. SBOM se najpogosteje izmenjuje v standardiziranih formatih, kot sta SPDX (Software Package Data Exchange) in CycloneDX, ki omogočata strojno obdelavo podatkov o komponentah ter njihovo vključevanje v avtomatizirane varnostne procese.

Pomen SBOM se pokaže predvsem pri odzivanju na varnostne incidente. Če je določena knjižnica ogrožena ali je v njej odkrita ranljivost, lahko organizacija s pomočjo SBOM preveri, ali je komponenta vključena v njene aplikacije. To vključuje tudi posredne odvisnosti (angl. transitive dependencies), ki jih organizacija morda ni neposredno izbrala, vendar so bile vključene prek drugih paketov. Brez takšnega pregleda je lahko nevarna komponenta prisotna v okolju dalj časa, saj organizacija nima celotnega pregleda nad uporabljenimi odvisnostmi.

Upravljanje odvisnosti (angl. dependency management) zato ne pomeni samo posodabljanja paketov, temveč vključuje tudi spremljanje njihovega izvora, preverjanje sprememb in nadzor nad tem, katere komponente so dovoljene v razvojnem procesu. Poseben izziv predstavljajo že prej omenjene prehodne odvisnosti (angl. transitive dependencies), kjer aplikacija neposredno ne uporablja določene knjižnice, vendar jo vključuje druga uporabljena komponenta [45], [46].

Pomembno vlogo pri tem imajo upravljalniki paketov (angl. package managers), saj predstavljajo eno izmed prvih mest, kjer se zunanja koda prenese in vključi v razvojno okolje. Ker lahko nekateri paketi med namestitvijo izvajajo dodatno kodo, na primer namestitvene skripte (angl. install scripts), ki se samodejno zaženejo ob namestitvi ali posodobitvi paketa, predstavlja sam proces namestitve pomembno varnostno točko.

Sodobni paketni ekosistemi zato uvajajo dodatne zaščitne mehanizme. Upravljalnik paketov npm omogoča preverjanje integritete paketov prek zaklenjenih datotek (angl. lock files) in kontrolnih vsot. Takšna datoteka zabeleži točne različice vseh uporabljenih paketov v projektu in njihovih odvisnosti, vključno s prehodnimi odvisnostmi (angl. transitive dependencies). Pri naslednji namestitvi se uporabijo enake različice, namesto da bi upravljalnik ponovno izbiral najnovejše razpoložljive pakete. Poleg tega vsebuje integritetne vrednosti (angl. integrity hashes) prenesenih paketov, ki jih upravljalnik paketov uporabi pri namestitvi za preverjanje, ali prenesena vsebina ustreza pričakovani vsebini. Če se vsebina paketa razlikuje od pričakovane vrednosti, lahko upravljalnik zazna nepričakovano spremembo in namestitev prepreči. Upravljalnik npm omogoča tudi preverjanje znanih ranljivosti z ukazom npm audit ter možnost onemogočanja samodejnega izvajanja namestitvenih skript. Takšne omejitve zmanjšujejo tveganje, da bi ogrožen paket med namestitvijo izvedel zlonamerno kodo [47].

Podobne pristope uvajajo tudi drugi upravljalniki paketov. Upravljalnik paketov pnpm uporablja zaklenjeno datoteko (angl. lock file) pnpm-lock.yaml, ki podobno kot pri npm zabeleži točne različice neposrednih in prehodnih odvisnosti ter zagotavlja ponovljive namestitve (angl. reproducible installs). Poleg tega pnpm omogoča dodatne zaščitne mehanizme, kot je omejevanje izvajanja namestitvenih skript in zmanjševanje tveganja zaradi na novo objavljenih zlonamernih paketov z mehanizmom časovnega zamika pri novih izdajah (angl. minimum release age) [48]. Namen teh mehanizmov je ustvariti časovno okno, v katerem lahko skupnost ali varnostni sistemi zaznajo sumljive spremembe, preden se paket samodejno razširi v večje število okolij.

Tudi Yarn vključuje več mehanizmov za povečanje varnosti programske dobavne verige. Podobno kot npm in pnpm uporablja zaklenjeno datoteko (angl. lock file), yarn.lock, ki določa natančne različice uporabljenih paketov in njihovih odvisnosti ter zagotavlja ponovljive namestitve (angl. reproducible installs). Yarn preverja integriteto

prenesenih paketov in s tem zmanjšuje tveganje nepričakovanih sprememb vsebine med namestitvijo. Novejše različice Yarn uvajajo dodatne zaščitne mehanizme, kot je omejevanje izvajanja namestitvenih skript ter strožji nadzor nad tem, kateri paketi lahko izvajajo dodatno kodo med namestitvijo.

V okolju Bun je varnost namestitvenega procesa obravnavana predvsem skozi nadzor izvajanja namestitvenih skript. Upravljalnik paketov bun paketov, ki med namestitvijo izvajajo dodatno kodo, ne obravnava samodejno kot zaupanja vrednih, temveč zahteva njihovo izrecno dovoljenje prek mehanizma zaupanja vrednih odvisnosti (angl. trusted dependencies). Tak pristop zmanjšuje tveganje, da bi ogrožen paket med namestitvijo samodejno izvedel zlonamerno kodo v razvijalčevem okolju ali procesu CI/CD [49].

Pri uvajanju teh zaščitnih mehanizmov je treba upoštevati tudi vpliv na razvojni proces. Varnostni ukrepi pogosto vključujejo kompromis med zmanjšanjem tveganja programske dobavne verige in ohranjanjem hitrosti ter prilagodljivosti razvoja. Omejevanje namestitvenih skript lahko na primer vpliva na pakete, ki jih legitimno uporabljajo za prevajanje izvorne kode ali pripravo okolja, zato je primernejši nadzorovan pristop z izrecnim dovoljevanjem preverjenih paketov. Podobno lahko uporaba nespremenljivih različic paketov ali slik zmanjša tveganje nepričakovanih sprememb, vendar zahteva dodatne postopke za pravočasno posodabljanje zaradi varnostnih popravkov. Namen zaščitnih ukrepov zato ni popolna odstranitev avtomatizacije ali prilagodljivosti, temveč vzpostavitev ravnotežja med zmanjšanjem napadalne površine in učinkovitim razvojnim procesom.

### *6.1.2. Preverjanje izvora in pristnosti artefaktov*

Pri sodobnem razvoju programske opreme ni dovolj preveriti samo končnega artefakta, temveč tudi način, kako je ta nastal. Programska oprema se danes pogosto gradi avtomatizirano v okoljih CI/CD, kjer izvorna koda, odvisnosti, orodja za gradnjo programske opreme in dostopna dovoljenja vplivajo na končni rezultat. Če napadalec pridobi dostop do enega izmed teh elementov, lahko ustvari legitimno videti artefakt, ki vsebuje zlonamerne spremembe.

Zato organizacije uvajajo preverjanje izvora programske opreme (angl. software provenance). Gre za informacije o poreklu programske opreme, ki opisujejo, kako je bil določen programski artefakt ustvarjen. Ti podatki lahko vključujejo povezavo do izvorne kode, uporabljeno različico repozitorija, uporabljeno okolje za gradnjo, uporabljena orodja ter informacije o tem, kateri proces CI/CD je ustvaril artefakt.

Primer takšnega artefakta je lahko npm paket, Docker slika ali prevedena aplikacija. Če organizacija prejme novo različico paketa, lahko s pomočjo provenance preveri, ali je bila ustvarjena iz pričakovane izvorne kode in prek zaupanja vrednega procesa gradnje. Brez takšnih informacij je težko razlikovati med legitimno novo različico in različico, ki jo je ustvaril napadalec.

Tak napad se lahko zgodi na primer tako, da napadalec pridobi dostop do razvojnega okolja ali sistema CI/CD in spremeni proces gradnje. Namesto da bi neposredno spremenil izvorno kodo, lahko spremeni avtomatiziran proces tako, da pri gradnji vstavi dodatno kodo. Rezultat je artefakt, ki je videti kot uradna izdaja, vendar vsebuje zlonamerno funkcionalnost. Posledice so lahko kraja skrivnosti, skrita zlonamerna funkcionalnost (angl. backdoor) ali ogrožanje vseh uporabnikov, ki prenesejo takšno različico.

Pomemben okvir na tem področju predstavlja SLSA. SLSA je nabor priporočil in zahtev za izboljšanje varnosti procesa izdelave programske opreme. Določa različne ravni zaščite, od osnovnega beleženja izvora artefaktov do naprednejših zahtev, kjer mora biti proces gradnje izoliran, preverljiv in zaščiten pred nepooblaščenimi spremembami. Namen SLSA je zmanjšati možnost, da bi napadalec neopazno vplival na proces med izvorno kodo in končnim artefaktom [3]. Pri preverjanju izvora in zaporedja korakov izdelave artefaktov se pogosto uporablja tudi model in-toto, ki omogoča beleženje in preverjanje posameznih dejanj v verigi nastanka programske opreme. SLSA se na te koncepte navezuje pri zagotavljanju preverljivega izvora artefaktov.

Dodatno zaščito omogoča podpisovanje artefaktov. Pri tem se programski artefakt, kot so paket npm, Docker slika, izvršna datoteka ali druga izdaja programske opreme, opremi s kriptografskim podpisom. Uporabnik ali

avtomatiziran sistem lahko nato preveri, ali je artefakt ustvaril pričakovani izdajatelj in ali se njegova vsebina po podpisu ni spremenila.

Tradicionalno je bilo podpisovanje pogosto odvisno od dolgoročnih zasebnih ključev, ki jih je bilo treba varno hraniti. Rešitve, kot je Sigstore, uvajajo drugačen pristop, kjer se uporabljajo kratkožive identitete (angl. short-lived identities) in javni zapisi podpisov. Namesto upravljanja dolgoročnih ključev lahko organizacija preveri povezavo med identiteto izvajalca gradnje v procesu CI/CD, procesom gradnje in podpisanim artefaktom [50].

### *6.1.3. Varovanje okolij CI/CD, skrivnosti in identitet*

Ker sistemi CI/CD predstavljajo povezavo med izvorno kodo, procesom gradnje programske opreme in produkcijskim okoljem, so ena izmed najpomembnejših točk zaščite v sodobni programski dobavni verigi. Za razliko od klasičnih razvojnih okolij imajo sistemi CI/CD pogosto dostop do različnih zunanjih storitev, registrov paketov, oblačnih okolij in produkcijskih sistemov. Če je CI/CD proces ogrožen, lahko napadalcu omogoči veliko več kot samo spremembo kode. Omogoči lahko krajo skrivnosti, dostop do infrastrukture ali vstavljanje zlonamernih sprememb v končne programske artefakte [51].

Napadi na okolja CI/CD pogosto ne izkoriščajo neposredno ranljivosti v aplikacijski kodi, temveč zlorabijo zaupanje, ki ga ima razvojni proces do različnih sistemov. Primer je lahko ogrožen delovni tok (angl. workflow) v GitHub Actions, GitLab CI ali podobnem sistemu, kjer napadalec spremeni ukaze, ki se izvajajo med gradnjo, ali zlorabi preširoka dovoljenja avtomatiziranega procesa. Ker se ti procesi izvajajo samodejno in pogosto z višjimi privilegiji, lahko ena sprememba vpliva na vse nadaljnje gradnje in objave programske opreme [51], [52].

Osnovni obrambni pristop je načelo najmanjših privilegijev (angl. least privilege). Vsak proces CI/CD, uporabnik ali avtomatizirana naloga naj ima samo tiste pravice, ki jih nujno potrebuje za svoje delovanje. Na primer, proces, ki samo gradi aplikacijo, ne bi smel imeti dovoljenj za spreminjanje izvorne kode ali dostopa do produkcijskih skrivnosti. Prav tako je pomembno ločevanje razvojnih, testnih in produkcijskih okolij, saj zloraba enega dela sistema ne sme samodejno omogočiti dostopa do vseh drugih [51], [53].

Pomemben del zaščite predstavlja upravljanje skrivnosti (angl. secrets management). Procesi CI/CD pogosto potrebujejo dostopne ključe, žetone in poverilnice za objavo paketov, dostop do oblakov ali povezavo z drugimi sistemi. Te informacije ne smejo biti shranjene v izvorni kodi ali neposredno v konfiguracijskih datotekah. Namesto tega se uporabljajo namenski sistemi za upravljanje skrivnosti, omejevanje dostopa in redna rotacija poverilnic. Posebej pomembno je tudi zmanjševanje življenjske dobe dostopnih podatkov, saj ukradena kratkotrajna poverilnica predstavlja manjše tveganje kot trajni dostopni ključ [51], [54].

Zaščititi je treba tudi samo konfiguracijo procesov CI/CD. Datoteke, ki definirajo delovne tokove, na primer GitHub Actions datoteke YAML, niso samo konfiguracija, temveč dejansko predstavljajo del razvojne infrastrukture. Sprememba ene vrstice lahko spremeni način gradnje, objave ali dostopa do zunanjih sistemov. Zato je pomembno, da so spremembe teh datotek predmet pregleda kode, omejenih dovoljenj in dodatnega nadzora. Pri zunanjih akcijah in razširitvah je priporočljivo preveriti njihov izvor ter uporabljati nespremenljive različice namesto dinamičnih oznak, ki lahko kasneje kažejo na drugo vsebino [51], [52].

Sodobni sistemi CI/CD zato postajajo podobni produkcijskim okoljem in jih je treba tudi varnostno obravnavati na enak način. Cilj ni popolnoma odstraniti avtomatizacije, temveč zagotoviti, da avtomatizirani procesi delujejo z omejenim zaupanjem in da uspešen napad na posamezno komponento ne omogoči celotnega prevzema programske dobavne verige [51], [53].

Posebno pozornost zahteva tudi zaščita izvajalnega okolja procesov CI/CD. Avtomatizirani izvajalci (angl. runners), ki izvajajo ukaze za gradnjo, testiranje in objavo programske opreme, pogosto delujejo z dostopom do izvorne kode, registrov paketov ali oblačnih storitev. Če napadalec pridobi nadzor nad takšnim okoljem, lahko zlorabi obstoječa dovoljenja za izvajanje ukazov, pridobivanje skrivnosti ali spreminjanje končnih artefaktov. Zato se sodobni pristopi osredotočajo tudi na izolacijo okolij za gradnjo, uporabo kratkotrajnih izvajalnih okolij, nadzor nad uporabljenimi orodji ter redno preverjanje konfiguracije avtomatiziranih procesov [51], [53].

### *6.1.4. Varnost kontejnerskih slik*

Kontejnerske slike (angl. container images) predstavljajo pomemben del sodobne programske dobavne verige, saj pogosto vsebujejo celotno okolje, potrebno za izvajanje aplikacije. Poleg aplikacijske kode lahko vključujejo sistemske knjižnice, zunanje pakete, konfiguracijo in druga orodja. Zaradi tega ogrožena kontejnerska slika ne vpliva samo na eno komponento, temveč lahko vnese zlonamerno kodo v vse sisteme, ki jo uporabijo.

Zaščita kontejnerskih slik zato vključuje preverjanje uporabljenih komponent, skeniranje znanih ranljivosti, uporabo zaupanja vrednih osnovnih slik (angl. base images) ter preverjanje izvora in integritete slike pred uporabo. Pomembna praksa je tudi uporaba minimalnih osnovnih slik, ki vsebujejo samo nujne komponente, saj manjša količina programske opreme pomeni manjšo potencialno napadalno površino [52].

Za pregled kontejnerskih slik se uporabljajo specializirana orodja za varnostno analizo, ki preverjajo znane ranljivosti v sistemskih paketih, knjižnicah in drugih vključenih komponentah. Orodja, kot so Trivy, Grype ali podobne rešitve, se pogosto vključujejo neposredno v procese CI/CD, kjer preverijo sliko še pred objavo v register ali uporabo v produkcijskem okolju. Takšen pristop omogoča zgodnje odkrivanje težav in preprečuje, da bi ranljive ali sumljive slike postale del nadaljnje distribucije.

Poseben problem predstavljajo spremenljive oznake (angl. mutable tags). Če se pri namestitvi uporablja na primer oznaka latest, lahko ista oznaka v registru kasneje kaže na drugo vsebino kot ob prvotni uporabi. Posledično lahko razvojna ali produkcijska okolja nehote prenesejo novo, nepreverjeno različico slike. Zato se v varnostno občutljivih okoljih uporabljajo nespremenljive reference, na primer kriptografski povzetki (angl. digests), ki enolično določajo točno določeno različico slike [52].

Dodatno zaščito omogoča podpisovanje kontejnerskih slik, kjer se pred uporabo preveri, ali je sliko ustvaril zaupanja vreden izdajatelj in ali vsebina po objavi ni bila spremenjena. Rešitve, kot je Sigstore s svojim orodjem Cosign, omogočajo kriptografsko podpisovanje slik in preverjanje njihove pristnosti z zmanjšano potrebo po klasičnem dolgoročnem upravljanju ključev. S tem se zmanjša tveganje, da bi napadalec zamenjal legitimno sliko v registru ali med distribucijskim procesom.

Cilj teh ukrepov je zagotoviti, da kontejnerska slika, ki pride v produkcijsko okolje, ustreza preverjeni različici, vsebuje samo pričakovane komponente in je bila ustvarjena skozi nadzorovan ter preverljiv proces.

## 6.2. Organizacijski ukrepi

Tehnični mehanizmi, predstavljeni v prejšnjem poglavju, predstavljajo pomemben del zaščite programske dobavne verige, vendar sami po sebi ne zadostujejo. Dobavna veriga vključuje tudi ljudi, razvojne procese, zunanje ponudnike in organizacijske odločitve, zato mora varnost temeljiti na kombinaciji tehničnih kontrol in ustrezno določenih postopkov. Cilj organizacijskih ukrepov je zmanjšati verjetnost napak, omejiti posledice zlorabe ter zagotoviti učinkovit odziv, ko do incidenta pride [55], [56].

Eden izmed osnovnih principov je načelo najmanjših privilegijev (angl. least privilege), ki določa, da uporabniki, razvijalci in avtomatizirani procesi ne smejo imeti več dostopa, kot ga nujno potrebujejo za svoje delo. Kot je bilo predstavljeno v prejšnjih poglavjih, lahko ogrožen razvijalski račun ali proces CI/CD predstavlja pomembno vstopno točko v programsko dobavno verigo. Zato morajo organizacije jasno določiti dostopne pravice, ločiti odgovornosti med različnimi vlogami ter redno preverjati, ali so dodeljena dovoljenja še vedno potrebna [55], [57].

Pomemben del varnosti predstavlja tudi vključevanje varnostnih pregledov v razvojni proces. Varnost ne sme biti obravnavana kot zadnji korak pred objavo programske opreme, temveč kot del celotnega življenjskega cikla razvoja. To vključuje določene postopke pregleda sprememb, odgovornosti razvijalcev, potrjevanje kritičnih sprememb ter opredeljene kriterije, kdaj je programsko opremo mogoče objaviti ali namestiti v produkcijo. Tehnična preverjanja, kot so skeniranje odvisnosti in analiza artefaktov, predstavljena v prejšnjem poglavju, so učinkovitejša, kadar so vključena v jasno določen razvojni proces [55], [57].

Poseben izziv predstavlja upravljanje tveganj tretjih ponudnikov. Organizacije danes pogosto uporabljajo zunanje knjižnice, storitve v oblaku in programske rešitve drugih ponudnikov, zato varnost njihove dobavne verige ni odvisna samo od lastnih ukrepov. Pomembno je, da organizacija pred uporabo določenega ponudnika oceni njegova varnostna tveganja, določi pričakovane varnostne zahteve ter opredeli postopke sodelovanja v primeru incidenta. Tehnični pristopi, kot je pregled sestave programske opreme prek SBOM, lahko pri tem pomagajo, vendar morajo biti podprti z ustreznimi procesi in odgovornostmi [56], [58].

Spremljanje programske dobavne verige mora biti neprekinjen proces in ne zgolj enkratno preverjanje ob uvedbi programske opreme. Ker se komponente, ponudniki in grožnje s časom spreminjajo, mora organizacija ohranjati pregled nad kritičnimi deli svoje dobavne verige ter spremljati spremembe, ki bi lahko vplivale na varnost. Pomembno je tudi, da so določeni postopki obveščanja in odgovorne osebe, ki lahko ocenijo vpliv morebitnega varnostnega incidenta ter sprejmejo ustrezne ukrepe [55], [56].

Nenazadnje mora organizacija imeti pripravljen odziv na incidente. Napadi na programsko dobavno verigo pogosto zahtevajo drugačen odziv kot klasični varnostni incidenti, saj lahko prizadenejo zunanje komponente, razvojna okolja in že distribuirano programsko opremo. Učinkovit odziv vključuje hitro ugotavljanje obsega vpliva, preklic ogroženih dostopov, obveščanje prizadetih uporabnikov, odstranitev nevarnih različic ter ponovno vzpostavitev zaupanja v programske artefakte [56], [59].

Učinkovita zaščita programske dobavne verige zato ni samo vprašanje uporabe varnostnih orodij, temveč zahteva kombinacijo tehnologije, procesov in odgovornosti. Tehnični ukrepi zmanjšujejo možnost uspešnega napada, organizacijski ukrepi pa zagotavljajo, da organizacija tveganja pravočasno prepozna, obvladuje in se nanje ustrezno odzove [55], [56]. Organizacije pri vzpostavljanju takšnih praks pogosto uporabljajo uveljavljene okvire, kot je NIST Secure Software Development Framework (SSDF), ki pomaga sistematično vključiti varnostne prakse v razvojni življenjski cikel [1].

## 7 Razprava

Analiza predstavljenih napadalnih vektorjev in incidentov kaže, da se varnost programskih dobavnih verig ne more več obravnavati kot omejen problem upravljanja zunanjih knjižnic. Programska rešitev danes nastane v porazdeljenem sistemu, ki vključuje repozitorije izvorne kode, javne in zasebne registre paketov, sisteme za gradnjo programske opreme, okolja CI/CD, oblačne storitve, kontejnerske registre, podpisne mehanizme ter identitete ljudi in avtomatiziranih procesov. Vsaka od teh komponent predstavlja samostojno napadalno površino, hkrati pa tudi povezavo zaupanja do drugih delov sistema. Posledično varnost končnega izdelka ni odvisna samo od kakovosti njegove izvorne kode, temveč od najšibkejše povezave v celotnem procesu njegovega nastanka in distribucije.

Takšna ugotovitev pojasnjuje premik od tradicionalnih napadov na posamezne komponente k sistemskim napadom na verige zaupanja. Zgodnji incidenti, kot sta event-stream in ua-parser-js, so pokazali posledice zlorabe široko uporabljene odvisnosti. SolarWinds in Codecov sta nato pokazala, da je mogoče z vplivanjem na proces gradnje ali distribucije doseči veliko število organizacij, ne da bi bilo treba neposredno napasti vsako izmed njih. Sodobne kampanje ta pristop dodatno razširjajo: ukradene poverilnice omogočijo dostop do repozitorija, spremenjen delovni tok razkrije nove skrivnosti, pridobljeni žetoni omogočijo objavo paketov, zlonamerni paket pa nato ogrozi nova razvojna okolja. Napad zato ni več enkratno dejanje, ampak proces, v katerem posamezen uspešen vdor ustvarja pogoje za naslednjo stopnjo širjenja.

Tradicionalni varnostni modeli pri takšnih napadih niso zadostni predvsem zato, ker so bili pogosto zasnovani okoli jasne meje med notranjim zaupanja vrednim okoljem in zunanjim nezaupanja vrednim omrežjem. V sodobnem razvoju ta meja ni jasno definirana. Koda iz javnega registra se izvaja v notranjem okolju CI/CD, zunanja storitev sodeluje pri preverjanju identitete, oblačni sistem izdaja začasne poverilnice avtomatiziranemu procesu, končni artefakt pa je lahko sestavljen iz več sto neposrednih in posrednih odvisnosti. Po začetnem vdoru

napadalec pogosto uporablja legitimne račune, podpisane objave in dovoljene komunikacijske poti. Klasični nadzor omrežnega prometa ali iskanje znanih vzorcev zlonamerne kode zato ne zazna nujno dejstva, da je bilo zlorabljeno samo razmerje zaupanja.

Pomembna posledica je, da posamezen varnostni ukrep ne zagotavlja celovite zaščite. Večfaktorska avtentikacija zmanjša nevarnost kraje gesel, vendar ne prepreči zlorabe že izdanega žetona, ogrožene seje ali zlonamernega delovnega toka. SBOM izboljša pregled nad sestavo izdelka, vendar sama ne dokazuje, da so navedene komponente varne ali da artefakt med gradnjo ni bil spremenjen. Digitalni podpis potrjuje izvor podpisa in nespremenjenost artefakta po podpisu, ne pa nujno, ali je bil podpisni proces sam varen. Tudi kratkoživi žetoni OIDC zmanjšajo tveganje dolgoročnih skrivnosti, vendar lahko napačno določena pravila zaupanja omogočijo izdajo veljavnih poverilnic nepooblaščenemu procesu. Ti primeri kažejo, da je treba varnostne kontrole obravnavati kot medsebojno povezane sloje in ne kot neodvisne rešitve.

Okviri, kot sta NIST SSDF in SLSA, zato poudarjajo procesne in dokazljive lastnosti razvoja: zaščito kode in razvojnih okolij, nadzor sprememb, ločevanje odgovornosti, sledljivost izvora ter preverljivo poreklo artefaktov (angl. provenance) [1], [3]. Njihova vrednost ni zgolj v uvedbi dodatnih orodij, temveč v oblikovanju odgovora na vprašanje, pod katerimi pogoji je mogoče določenemu artefaktu zaupati. Varnostna odločitev se tako premakne od predpostavke, da je paket varen zato, ker prihaja iz znanega registra, k preverjanju njegovega izvora, postopka gradnje, uporabljenih identitet in pravil, po katerih je bil odobren za uporabo.

Hkrati ugotovitve kažejo, da odgovornosti ni mogoče prenesti samo na vzdrževalce odprtokodnih projektov. Številne kritične komponente vzdržuje majhno število posameznikov, medtem ko njihove pakete uporabljajo velike organizacije z bistveno večjimi finančnimi in kadrovskimi zmogljivostmi. Takšno nesorazmerje ustvarja sistemsko tveganje: ekosistem je odvisen od projektov, katerih vzdrževanje, upravljanje dostopov in odzivanje na incidente pogosto temelji na prostovoljnem delu. Organizacije morajo zato poleg preverjanja odvisnosti ocenjevati tudi stanje projektov, od katerih so odvisne, zmanjševati nepotrebno število komponent, podpirati ključne vzdrževalce ter pripravljati nadomestne možnosti za primere, ko projekt ni več ustrezno vzdrževan.

Umetna inteligenca to dinamiko dodatno okrepi, vendar je ne spremeni v povsem nov varnostni problem. Kot je bilo predstavljeno v petem poglavju, umetna inteligenca povečuje količino ustvarjene kode, hitrost vključevanja sprememb in stopnjo avtomatizacije razvojnih opravil. S tem se povečuje razkorak med hitrostjo nastajanja sprememb in zmožnostjo njihovega kakovostnega preverjanja. Pri agentnih sistemih je učinek še izrazitejši, saj model ne pripravlja več samo predlogov, temveč lahko namešča odvisnosti, spreminja konfiguracije in uporablja razvojne identitete. NIST-ov profil SSDF za generativno umetno inteligenco potrjuje, da morajo biti UI-komponente vključene v obstoječe postopke varnega razvoja, upravljanja dobaviteljev, dokumentiranja in preverjanja, ne pa obravnavane kot ločen tehnološki dodatek [1].

Iz tega izhaja potreba po novem modelu človeškega nadzora. Zahteva, da človek formalno potrdi vsako spremembo, ni učinkovita, če pregledovalec zaradi obsega ali kompleksnosti spremembe ne razume njenega učinka. Smiselni nadzor mora biti usmerjen v tveganja: avtomatizirane spremembe z dostopom do novih odvisnosti, skrivnosti, produkcijskih dovoljenj ali postopkov objave morajo zahtevati strožji pregled kot lokalne in omejene spremembe. Pomembna je tudi ločitev privilegijev, pri kateri razvojni agent, proces CI/CD ali posamezni vzdrževalec nima samostojnega dostopa do vseh korakov od spremembe kode do objave v produkcijo.

Varnost dobavnih verig ima tudi izrazito medorganizacijsko razsežnost. Organizacija lahko dobro zaščiti lastno infrastrukturo, vendar ostane izpostavljena prek ponudnika programske opreme, upravljane storitve ali odprtokodne komponente. ENISA je že pri analizi incidentov med letoma 2020 in 2021 ugotovila, da so napadalci pogosto napadli dobavitelja, da bi dosegli njegove stranke, poznejše analize pa še naprej izpostavljajo digitalno dobavno verigo kot pomemben vektor napadov [2]. Zato ocenjevanje dobaviteljev ne sme biti omejeno na vprašalnike in pogodbene izjave, ampak mora vključevati konkretne dokaze o načinu razvoja, upravljanju ranljivosti, izvoru artefaktov, zaščiti identitet ter sposobnosti obveščanja in odzivanja ob incidentu.

Na podlagi obravnavanih primerov je mogoče programske dobavne verige razumeti kot kompleksen sociotehnični sistem. Tehnični elementi, kot so paketi, kontejnerske slike in žetoni, so povezani z organizacijskimi odločitvami, ekonomskimi omejitvami odprtokodnega razvoja in človeškim zaupanjem. Napadalci lahko prehajajo med temi ravnmi: s socialnim inženiringom pridobijo račun, z računom spremenijo avtomatizirani proces, prek procesa pridobijo identiteto, z identiteto pa objavijo tehnično veljaven artefakt. Obramba mora zato povezovati tehnične kontrole, upravljanje identitet, razvojne politike, nadzor dobaviteljev in odzivanje na incidente. Načela CISA Secure by Design podobno poudarjajo, da mora odgovornost za varnost postati sestavni del načrtovanja, izdelave in vzdrževanja programske opreme, ne pa breme, ki se pretežno prenaša na končnega uporabnika [60].

Obravnava ima tudi nekatere omejitve. Področje se hitro razvija, poimenovanja kampanj in začetne tehnične analize incidentov pa se lahko med viri razlikujejo. Pri novejših incidentih so pogosto najprej dostopna poročila varnostnih podjetij in upravljavcev prizadetih platform, medtem ko neodvisne akademske analize nastanejo pozneje. Prispevek zato predvsem sistematizira ponavljajoče se napadalne vzorce in odnose zaupanja, ne pa izčrpno vseh javno znanih incidentov. Nadaljnje raziskave bi morale bolj sistematično meriti dejanski učinek posameznih obrambnih ukrepov, proučiti prehodne oziroma posredne odvisnosti med razvojnimi storitvami ter oblikovati metode za ocenjevanje tveganja razvojnih agentov, ki delujejo z različnimi stopnjami avtonomije.

Skupna ugotovitev razprave je, da cilj varnosti ne more biti popolna odprava zaupanja, saj sodobni razvoj temelji prav na sodelovanju in ponovni uporabi komponent. Cilj mora biti zmanjšanje implicitnega zaupanja ter njegova zamenjava z omejenim, preverljivim in prekličljivim zaupanjem. Vsaka identiteta naj ima najmanjše potrebne privilegije, vsak pomemben artefakt preverljiv izvor, vsak korak objave sledljiv zapis, vsaka zunanja komponenta pa jasno opredeljeno odgovornost in možnost odziva. Takšen pristop ne prepreči vseh kompromitacij, vendar omeji njihovo širjenje in omogoči hitrejše razumevanje, katere povezave v dobavni verigi so bile prizadete.

## 8 Zaključek

Programske dobavne verige so se iz podpornega elementa razvoja programske opreme preoblikovale v eno ključnih področij sistemskega kibernetskega tveganja. Analiza razvoja napadov kaže, da sodobni napadalci vse redkeje ciljajo samo posamezno ranljivost ali komponento. Namesto tega izkoriščajo medsebojno povezanost odprtokodnih odvisnosti, računov vzdrževalcev, repozitorijev, okolij za gradnjo programske opreme, procesov CI/CD, registrov artefaktov, identitetnih sistemov in oblačne infrastrukture. Uspešen napad na eno samo komponento lahko zato sproži verigo nadaljnjih zlorab ter vpliva na veliko število organizacij, ki si neposredno ali posredno delijo isto komponento oziroma ponudnika.

Osrednja ugotovitev prispevka je, da sodobnih napadov na programske dobavne verige ni več ustrezno obravnavati kot posamezne nepovezane tehnične incidente. Gre za večstopenjske in pogosto avtomatizirane operacije, pri katerih napadalci postopoma prevzemajo legitimne identitete, procese in distribucijske kanale. Zaradi tega lahko zlonamerna dejavnost navzven deluje kot običajen razvojni postopek ali postopek objave. Najpomembnejša tarča tako ni nujno sama programska koda, temveč zaupanje, ki povezuje razvijalce, avtomatizirane procese, zunanje ponudnike in končne uporabnike.

Umetna inteligenca tega problema ne ustvarja sama, vendar ga lahko pomembno pospeši. Orodja, temelječa na velikih jezikovnih modelih, povečujejo hitrost ustvarjanja kode, testov, konfiguracij in avtomatiziranih sprememb, hkrati pa lahko zmanjšajo delež neposrednega človeškega preverjanja. Halucinirane odvisnosti, sprejemanje nepreverjene generirane kode, posredni napadi z vbrizgavanjem navodil ter široke pravice razvojnih agentov širijo napadalno površino. Na strani napadalcev lahko enaka orodja pospešijo iskanje ranljivih komponent, pripravo prepričljivega socialnega inženiringa, prilagajanje zlonamerne kode in izvajanje obsežnejših kampanj.

Učinkovita zaščita zato zahteva večplastni in sistemski pristop. Organizacije morajo poleg upravljanja odvisnosti in izdelave seznama programskih komponent preverjati izvor ter celovitost artefaktov, omejevati privilegije razvojnih in identitetnih sistemov, varovati skrivnosti, utrjevati okolja CI/CD ter spremljati spremembe po celotni

poti od izvorne kode do produkcije. Tehnični ukrepi morajo biti dopolnjeni z organizacijskimi procesi, ki vključujejo nadzor tretjih ponudnikov, jasno razmejitev odgovornosti, pripravljenost na preklic kompromitiranih poverilnic in artefaktov ter usklajen odziv na incidente, ki lahko presegajo meje posamezne organizacije.

Prihodnje raziskave bi morale podrobneje preučiti merjenje kaskadnega tveganja med povezanimi organizacijami, samodejno odkrivanje zlorab v CI/CD in federiranih identitetnih okoljih ter preverjanje porekla kode in odločitev, ki jih ustvarjajo razvojni agenti. Posebno pomembni bodo modeli nadzora, ki omogočajo uporabo umetne inteligence brez opustitve preverljivosti, načela najmanjših privilegijev in odgovornega človeškega odločanja. Varnost programske dobavne verige bo v prihodnje zato odvisna predvsem od sposobnosti organizacij, da ne varujejo le posameznih komponent, temveč neprekinjeno preverjajo celotno verigo zaupanja, skozi katero programska oprema nastane, se distribuira in izvaja.

## Literatura

[1] M. Souppaya, K. Scarfone, and D. Dodson, "Secure Software Development Framework (SSDF) version 1.1 :," Feb. 2022. doi: 10.6028/NIST.SP.800-218.

[2] Ifigeneia. Lella, Marianthi. Theocharidou, Eleni. Tsekmezoglou, Apostolos. Malatras, S. García, and Veronica. Valeros, *ENISA threat landscape for supply chain attacks*. ENISA, 2021.

[3] SLSA Framework, "SLSA: Supply-chain Levels for Software Artifacts." Accessed: Aug. 01, 2026. [Online]. Available: https://github.com/slsa-framework/slsa

[4] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, doi: 10.1145/358198.358210.

[5] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," 2020, pp. 23–43. doi: 10.1007/978-3-030-52683-2_2.

[6] "Active Exploitation of SolarWinds Software | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/news-events/alerts/2020/12/13/active-exploitation-solarwinds-software

[7] "Bash Uploader Security Update - Codecov." Accessed: Aug. 01, 2026. [Online]. Available: https://about.codecov.io/security-update/

[8] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. De Carli, "Beyond Typosquatting: An In-depth Look at Package Confusion," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 3439–3456. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/neupane

[9] A. Freund, "oss-security - backdoor in upstream xz/liblzma leading to ssh server compromise." Accessed: Aug. 01, 2026. [Online]. Available: https://www.openwall.com/lists/oss-security/2024/03/29/4

[10] R. Cox, "research!rsc: Timeline of the xz open source attack." Accessed: Aug. 01, 2026. [Online]. Available: https://research.swtch.com/xz-timeline

[11] A. Hahami, "OH-MY-DC: OIDC Misconfigurations in CI/CD." Accessed: Aug. 01, 2026. [Online]. Available: https://unit42.paloaltonetworks.com/oidc-misconfigurations-in-ci-cd/

[12] Imperva, "What is Dependency Confusion | Risks & Prevention | Imperva." Accessed: Aug. 01, 2026. [Online]. Available: https://www.imperva.com/learn/application-security/dependency-confusion/

[13] A. Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies | by Alex Birsan | Medium." Accessed: Aug. 01, 2026. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[14] Endor Labs, "Dependency Confusion: How Attackers Poison Your Build | Blog | Endor Labs." Accessed: Aug. 01, 2026. [Online]. Available: https://www.endorlabs.com/learn/dependency-confusion-how-attackers-poison-your-build

[15] Rescana, "In-Depth Analysis: Supply Chain Poisoning of Popular npm Packages Exploiting event-stream, ua-parser-js, and More – Rescana." Accessed: Aug. 01, 2026. [Online]. Available: https://www.rescana.com/post/in-depth-analysis-supply-chain-poisoning-of-popular-npm-packages-exploiting-event-stream-ua-parser

[16] npm Blog, "npm Blog Archive: Details about the event-stream incident." Accessed: Aug. 01, 2026. [Online]. Available: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

[17] P. Muncaster, "What is Typosquatting? Definition, Attack Types & Prevention." Accessed: Aug. 01, 2026. [Online]. Available: https://www.eset.com/blog/en/home-topics/privacy-and-identity-protection/what-is-typosquatting-how-do-i-tackle-it/

[18] A. Stiefel and J. Gile, "Major Supply Chain Attack Compromises Popular npm Packages Including chalk and debug | Blog | Endor Labs." Accessed: Aug. 01, 2026. [Online]. Available: https://www.endorlabs.com/learn/major-supply-chain-attack-compromises-popular-npm-packages-including-chalk-and-debug

[19] J. F. Bonaobra and J. Aquino, "What We Know About the NPM Supply Chain Attack | Trend Micro (US)." Accessed: Aug. 01, 2026. [Online]. Available: https://www.trendmicro.com/en_us/research/25/i/npm-supply-chain-attack.html

[20] M. Fiedler, "Token Exfiltration Campaign via GitHub Actions Workflows - The Python Package Index Blog." Accessed: Aug. 01, 2026. [Online]. Available: https://blog.pypi.org/posts/2025-09-16-github-actions-token-exfiltration/

[21] M. Abidi, C. Decotignie, and G. Plante, "CI/CD Security: Supply chain attack from a compromised developer - RiskInsight." Accessed: Aug. 01, 2026. [Online]. Available: https://www.riskinsight-wavestone.com/en/2026/07/ci-cd-security-supply-chain-attack-from-a-compromised-developer/

[22] CyberSierra, "Supply Chain Attacks in CI/CD Pipelines: The New Cloud Security Nightmare." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cybersierra.co/blog/prevent-cicd-supply-chain-attacks

[23] M. Ayenson, "CI/CD pipeline abuse: the problem no one is watching — Elastic Security Labs." Accessed: Aug. 01, 2026. [Online]. Available: https://www.elastic.co/security-labs/detecting-cicd-pipeline-abuse-with-llm-augmented-analysis

[24] R. Nisimi, "TeamPCP Exposes Checkmarx CI/CD Secrets | Orca Security." Accessed: Aug. 01, 2026. [Online]. Available: https://orca.security/resources/blog/checkmarx-supply-chain-compromise-ci-cd-secrets/

[25] A. Kurmi, "5 Supply Chain Attacks in 48 Hours: Why Securing One Layer Is Not Enough - StepSecurity." Accessed: Aug. 01, 2026. [Online]. Available: https://www.stepsecurity.io/blog/5-supply-chain-attacks-in-48-hours-why-securing-one-layer-is-not-enough

[26] M. Shubel, "Cordyceps flaw pattern is more proof CI/CD is part of the attack surface - The New Stack." Accessed: Aug. 01, 2026. [Online]. Available: https://thenewstack.io/cordyceps-cicd/

[27] T. Linsley, "Postmortem: TanStack npm supply-chain compromise | TanStack Blog." Accessed: Aug. 01, 2026. [Online]. Available: https://tanstack.com/blog/npm-supply-chain-compromise-postmortem

[28] Gerrard Sarah, C. Crutchley, J. Herrington, Linsey Tanner, and Pellet Florian, "Hardening TanStack After the npm Compromise | TanStack Blog." Accessed: Aug. 01, 2026. [Online]. Available: https://tanstack.com/blog/incident-followup

[29] Ossprey, "Megalodon: Active GitHub Actions Supply Chain Attack." Accessed: Aug. 01, 2026. [Online]. Available: https://www.ossprey.com/blog/megalodon-credential-harvester-attack

[30] The Chainguard Team, "Container Hardening: Securing your software supply chain." Accessed: Aug. 01, 2026. [Online]. Available: https://www.chainguard.dev/supply-chain-security-101/container-hardening-securing-your-software-supply-chain

[31] A. Morag, "Supply Chain Attacks Using Container Images - Aqua." Accessed: Aug. 01, 2026. [Online]. Available: https://www.aquasec.com/blog/supply-chain-threats-using-container-images/

[32] S. Chierici, "Analysis on Docker Hub malicious images: Attacks through public container images | Sysdig." Accessed: Aug. 01, 2026. [Online]. Available: https://www.sysdig.com/blog/analysis-of-supply-chain-attacks-through-public-docker-images

[33] "Trivy, KICS, and the shape of supply chain attacks so far in 2026 | Docker." Accessed: Aug. 01, 2026. [Online]. Available: https://www.docker.com/blog/trivy-kics-and-the-shape-of-supply-chain-attacks-so-far-in-2026/

[34] Santos Jacob and J. R. Navato, "Analyzing TeamPCP's Supply Chain Attacks: Checkmarx KICS and elementary-data in CI/CD Credential Theft | Trend Micro (US)." Accessed: Aug. 01, 2026. [Online]. Available: https://www.trendmicro.com/en_us/research/26/e/analyzing-teampcp-supply-chain-attacks.html

[35] B. Clark, "npm Supply Chain Attack via Open Source maintainer compromise | Snyk." Accessed: Aug. 01, 2026. [Online]. Available: https://snyk.io/blog/npm-supply-chain-attack-via-open-source-maintainer-compromise/

[36] J. F, "Software supply chain attacks: check your dependencies | National Cyber Security Centre." Accessed: Aug. 01, 2026. [Online]. Available: https://www.ncsc.gov.uk/blogs/software-supply-chain-attacks-check-your-dependencies

[37] "Artificial intelligence risk management framework :," Jul. 2024. doi: 10.6028/NIST.AI.600-1.
[38] "Secure Software Development Framework | CSRC." Accessed: Aug. 01, 2026. [Online]. Available: https://csrc.nist.gov/projects/ssdf
[39] E. Kalliamvakou, "Research: quantifying GitHub Copilot's impact on developer productivity and happiness - The GitHub Blog." Accessed: Aug. 01, 2026. [Online]. Available: https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/
[40] P. Fontana, Y. Lee, H. Iyer, and S. Sharma, "2025 NIST GenAI (Pilot): Code Challenge Evaluation Plan | NIST." Accessed: Aug. 01, 2026. [Online]. Available: https://www.nist.gov/publications/2025-nist-genai-pilot-code-challenge-evaluation-plan
[41] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2022, pp. 754–768. doi: 10.1109/SP46214.2022.9833571.
[42] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the Security of GitHub Copilot Generated Code -- A Targeted Replication Study," Nov. 2023.
[43] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, and B. Viswanath, "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," in *34th USENIX Security Symposium (USENIX Security 25)*, Seattle, WA: USENIX Association, Aug. 2025, pp. 3687–3706. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen
[44] "AI Agent Security - OWASP Cheat Sheet Series." Accessed: Aug. 01, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/AI_Agent_Security_Cheat_Sheet.html
[45] "SBOM Resources Library | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/topics/information-communications-technology-supply-chain-security/sbom/resources-library
[46] "Software Supply Chain Security Guidance | NIST." Accessed: Aug. 01, 2026. [Online]. Available: https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-supply-chain-security-guidance
[47] "Documentation for the npm registry, website, and command-line interface." Accessed: Aug. 01, 2026. [Online]. Available: https://docs.npmjs.com/
[48] "Mitigating supply chain attacks | pnpm." Accessed: Aug. 01, 2026. [Online]. Available: https://pnpm.io/supply-chain-security
[49] "Add a trusted dependency - Bun." Accessed: Aug. 01, 2026. [Online]. Available: https://bun.com/docs/guides/install/trusted
[50] "Home · Sigstore." Accessed: Aug. 01, 2026. [Online]. Available: https://www.sigstore.dev/
[51] "CI CD Security - OWASP Cheat Sheet Series." Accessed: Aug. 01, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/CI_CD_Security_Cheat_Sheet.html
[52] M. Souppaya, J. Morello, and K. Scarfone, "SP 800-190, Application Container Security Guide | CSRC." Accessed: Aug. 01, 2026. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/190/final
[53] R. Chandramouli, F. Kautz, and S. Torres Arias, "SP 800-204D, Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD pipelines | CSRC." Accessed: Aug. 01, 2026. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/204/d/ipd
[54] "OpenID Connect - GitHub Docs." Accessed: Aug. 01, 2026. [Online]. Available: https://docs.github.com/en/actions/concepts/security/openid-connect
[55] "Software Supply Chain Security - OWASP Cheat Sheet Series." Accessed: Aug. 01, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Software_Supply_Chain_Security_Cheat_Sheet.html
[56] J. Boyens, A. Smith, N. Bartol, K. Winkler, A. Holbrook, and M. Fallon, "SP 800-161 Rev. 1, Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations | CSRC." Accessed: Aug. 01, 2026. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/161/r1/upd1/final
[57] "Securing the Software Supply Chain: Recommended Practices for Developers | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/resources-tools/resources/securing-software-supply-chain-recommended-practices-developers

[58] "Securing the Software Supply Chain: Recommended Practices Guide for Customers and accompanying Fact Sheet | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/resources-tools/resources/securing-software-supply-chain-recommended-practices-guide-customers-and

[59] "Defending Against Software Supply Chain Attacks | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/resources-tools/resources/defending-against-software-supply-chain-attacks

[60] "Secure by Design | CISA." Accessed: Aug. 01, 2026. [Online]. Available: https://www.cisa.gov/securebydesign