Asystenci kodujący i agenci w terminalu: jak pracować bezpiecznie

Poziom: średni

Dotyczy: asystenci kodujący (kategorie, bez wskazania produktu), Docker, Gitea 1.27

Czas: ~45 min

Programista dostaje dziś trzy rodzaje pomocy: podpowiedzi w edytorze, czat obok kodu i agenta, który sam czyta pliki, uruchamia polecenia i zapisuje zmiany. Pierwsze dwa mają zasięg jednej linii albo jednego pliku, trzeci ma zasięg wszystkiego, do czego dostał uprawnienia. Ta strona opisuje, jak ten trzeci rodzaj zamknąć w ramach, które zespół da się obronić przed audytorem i przed własną pomyłką.

Strona nie poleca żadnego produktu: zasady są takie same dla każdego narzędzia z tej kategorii. Komendy wykonujemy z CLI bash na stacji roboczej z Linuksem, z zainstalowanym Dockerem i Gitem. Podstawy uprawnień agentów opisuje strona Uprawnienia, prompt injection i zasięg rażenia, a tutaj przekładamy je na codzienną pracę z kodem.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
CI	Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie; w ITIL także configuration item, czyli element konfiguracji: każdy składnik usługi zapisany w CMDB
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
LLM	Large Language Model: duży model językowy, program przewidujący kolejne słowa tekstu
OWASP	Open Worldwide Application Security Project: organizacja publikująca listy najczęstszych zagrożeń, także dla aplikacji z AI

Fiszka

◫ OSTROŻNIE · NIEODWRACALNE

Uruchomienie poleceń w izolowanym kontenerze: docker run

```
docker run --rm -it \
  --network none \
  --cap-drop ALL \
  --security-opt no-new-privileges \
  --read-only --tmpfs /tmp \
  --pids-limit 256 --memory 4g \
  --user 1000:1000 \
  -v "$PWD":/work -w /work \
  NazwaObrazu bash
```

(gdzie `NazwaObrazu` to obraz z narzędziami do budowy i testów projektu, przygotowany wcześniej razem z zależnościami, bo kontener nie ma sieci)

Kontener widzi tylko katalog projektu, nie ma sieci, nie ma uprawnień roota i nie może zapisywać poza `/work` i `/tmp`. W takim kontenerze uruchamiamy testy i polecenia budowania. Przełączniki sprawdzamy w dokumentacji Dockera dla zainstalowanej wersji.

> Dostęp do gniazda `/var/run/docker.sock` jest równoznaczny z uprawnieniami roota na hoście, o czym piszemy na stronie o Portainerze. Do kontenera z agentem nigdy go nie montujemy. Przy zwykłym Dockerze użytkownik w grupie `docker` ma te same prawa, więc na stacjach z agentami wybieramy Podmana bez roota albo osobną maszynę wirtualną.

◫ **docker** = narzędzie do kontenerów

◫ TYLKO ODCZYT

Utworzenie osobnej kopii roboczej dla agenta: git worktree

```
git worktree add ../NazwaRepozytorium-agent -b agent/NazwaZadania
```

(gdzie `agent/NazwaZadania` to nowa gałąź tylko dla tego zadania, a katalog obok repozytorium to druga kopia robocza tego samego repozytorium)

Agent pracuje w osobnym katalogu na osobnej gałęzi, więc nasze niezapisane zmiany go nie dotyczą, a jego zmiany nie wchodzą nam do pracy.

git = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

TYLKO ODCZYT

Przegląd zmian agenta: git diff

```
git status --short
git diff --stat
git diff
```

Najpierw patrzymy, które pliki się zmieniły i ile linii, a dopiero potem czytamy treść. Pliki spoza zadania są pierwszym sygnałem ostrzegawczym.

Przykład:

```
 M src/cennik/import_cen.py
 M tests/test_import_cen.py
 M .gitea/workflows/build.yaml
 src/cennik/import_cen.py         | 14 +++++++-------
 tests/test_import_cen.py         |  6 +++---
 .gitea/workflows/build.yaml      |  4 ++++
 3 files changed, 14 insertions(+), 10 deletions(-)
```

git = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

TYLKO ODCZYT

Sprawdzenie, czy agent dotykał testów i plików CI

```
git diff --name-only main...HEAD -- tests .gitea/workflows
```

Zmiana w testach oznacza, że agent mógł dopasować test do błędnego kodu, a zmiana w `.gitea/workflows` może zmienić to, co build robi z sekretami. Oba przypadki czytamy linia po linii.

CI = Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie; w ITIL także configuration item, czyli element konfiguracji: każdy składnik usługi zapisany w CMDB · **git** = system kontroli wersji (nazwa to brytyjski slang: nieznośny typ)

□□ TYLKO ODCZYT

Skanowanie repozytorium pod kątem sekretów: gitleaks

```
gitleaks dir
gitleaks git
```

Pierwsze polecenie skanuje pliki w bieżącym katalogu, drugie całą historię commitów. Narzędzie zgłasza hasła, tokeny i klucze rozpoznane po wzorcach. Opcje sprawdzamy w `gitleaks --help` dla zainstalowanej wersji.

□□ **gitleaks** = narzędzie szukające haseł, tokenów i kluczy w plikach i w historii repozytorium Git

□□ ZMIENIA SYSTEM

Skanowanie sekretów przed każdym commitem: pre-commit install

```
vim .pre-commit-config.yaml
pre-commit install
```

```
repos:
  - repo: https://github.com/gitleaks/gitleaks
    rev: v8.24.2
    hooks:
      - id: gitleaks
```

Numer w polu `rev` podnosimy do aktualnego wydania projektu. Hak działa na stacji każdego programisty i agenta, ale można go ominąć, dlatego ten sam skan uruchamiamy jeszcze w CI .

□□ **CI** = Continuous Integration: automatyczne budowanie i testowanie kodu po każdej zmianie; w ITIL także configuration item, czyli element konfiguracji: każdy składnik usługi zapisany w CMDB · **vim** = Vi IMproved: ulepszony edytor vi · **pre-commit** = framework uruchamiający kontrole, na przykład skan sekretów, automatycznie przed każdym commitem

□□ TYLKO ODCZYT

Wyłączenie plików z sekretami z repozytorium: .gitignore

```
.env
.env.*
```

```
*.pem
*.key
secrets/
```

Wpisujemy je do `.gitignore`. Agent, który widzi katalog projektu, czyta też pliki `.env`, więc najlepiej, gdy sekretów w katalogu projektu w ogóle nie ma, a program dostaje je ze zmiennych środowiskowych w miejscu uruchomienia.

KLIKAMY W INTERFEJSIE

Założenie konta i tokena dla agenta w Gitei

```
Konto agent-ola: zwykłe konto użytkownika, dodane jako współpracownik z prawem zapisu tylko do wybranych repozytoriów
Settings → Applications → Generate New Token: zakresy read:repository i write:repository (opcjonalnie read:issue)
```

Nie nadajemy zakresów `admin` ani `organization`. Konto agenta ma własną tożsamość, więc w historii commitów i w logu Gitei widać, co zrobił człowiek, a co agent. Więcej o tokenach na stronie Pierwsze repozytorium w Gitea.

TYLKO ODCZYT

Sprawdzenie, kim jest token i do czego ma dostęp

```
curl -s -H "Authorization: token SekretTokenu" https://git.zielona.example/api/v1/user | jq '.login'
curl -s -H "Authorization: token SekretTokenu" https://git.zielona.example/api/v1/user/repos | jq '.[].full_name'
```

Przykład:

```
"agent-ola"
"zespol-ola/import-cen"
"zespol-ola/sklep-b2b-integracja"
```

Lista repozytoriów ma być równa liście, którą świadomie przyznaliśmy. Jeśli zawiera coś więcej, odbieramy uprawnienia, zanim agent zacznie pracę.

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

TYLKO ODCZYT

Ochrona gałęzi main w Gitei

```
Repozytorium → Settings → Branches → Add Branch Protection Rule
Nazwa gałęzi: main
Disable Push (zmiany tylko przez pull request)
Required Approvals: 1
Enable Status Check (testy z CI muszą przejść)
```

Agent wysyła zmiany na swoją gałąź `agent/...` i otwiera pull request, a scala je człowiek.

Opis tych ustawień jest na stronie Pierwsze repozytorium w Gitea.

ZRZUT EKRANU DO UZUPEŁNIENIA

Zrzut do uzupełnienia: ekran Branch Protection Rule dla `main` w repozytorium `import-cen`, z zaznaczonym Disable Push, Required Approvals ustawionym na 1 i włączonym Status Check.

Z życia wzięte

Z ŻYCIA WZIĘTE

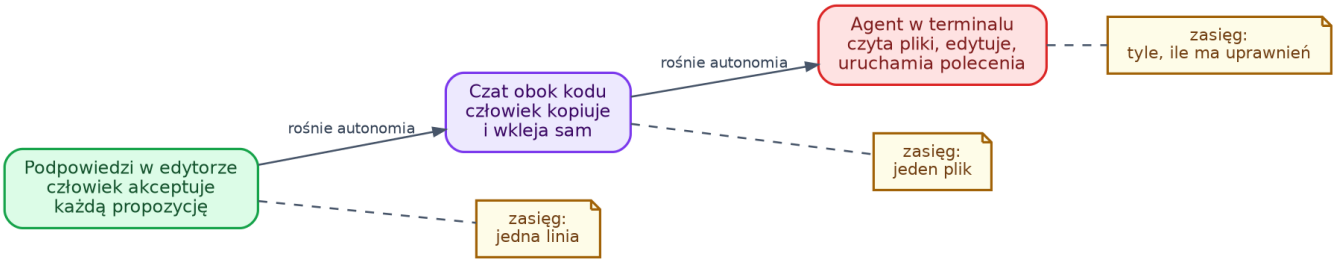
Środa, 10:40. Ola prosi agenta w terminalu o poprawkę importu cen do sklepu B2B: po zmianie formatu pliku jeden test przestał przechodzić. Agent „naprawia" go, luzując asercję w teście, i próbuje wysłać zmianę prosto na `main`, ale ochrona gałęzi odrzuca push. Otwiera więc pull request z gałęzi `agent/import-cen`, a Ola w diffie od razu widzi poluzowany test i cofa go. Prawdziwy błąd, źle zinterpretowaną kolumnę z ceną, poprawia sama w pięć minut.

Jak to działa

Rozwiń: jak to działa pod spodem

Trzy poziomy autonomii

> **Pojęcie: agent w terminalu.** Program, w którym model prowadzi pętlę opisaną na stronie Różnica pomiędzy czatem a agentem: czyta pliki projektu, edytuje je, uruchamia polecenia powłoki i testy, a często także commituje zmiany. Działa z uprawnieniami konta, na którym go uruchomiono.



Podpowiedzi w edytorze, czat obok kodu i agent w terminalu różnią się autonomią i zasięgiem skutków (kliknięcie otwiera pełny rozmiar)

Tabela zestawia kategorie narzędzi, bez wskazywania konkretnych produktów.

Kategoria	Co robi	Kto zatwierdza	Ryzyko	Podstawowa kontrola
Podpowiedzi w edytorze	dopisuje następną linię albo funkcję	człowiek, przyjmując każdą propozycję	błędna albo niebezpieczna linia w kodzie	zwykły przegląd kodu
Czat obok kodu	odpowiada na pytania, pisze fragmenty do wklejenia	człowiek, który wkleja	wklejone sekrety, błędny kod	zasady co wolno wklejać
Agent w terminalu albo edytorze	czyta pliki, uruchamia polecenia, zmienia kod	człowiek po fakcie (diff) albo przy każdym poleceniu	skutki poleceń, wyciek danych, zmiany poza zadaniem	piaskownica, uprawnienia, przegląd
Agent w chmurze albo w CI	wykonuje zadanie bez nas i zwraca pull request	człowiek przy scaleniu	jak wyżej, ale bez nadzoru w trakcie	wąski token, chroniony `main`, przegląd

Im wyżej w tabeli, tym mniej zależy od naszej uwagi w danej sekundzie, a im niżej, tym bardziej zależy od ustawień, które zrobiliśmy wcześniej. Dlatego agent wymaga innego podejścia niż podpowiedzi: podpowiedź sprawdzamy oczami, a agenta ograniczamy technicznie.

Przegląd zmian

Zmiany agenta czytamy tak, jak czytamy pull request od nowej osoby w zespole, a odpowiedzialność za scalony kod zostaje po stronie człowieka. Zadania dajemy małe, jedno na gałąź, bo duży diff robi się nieczytelny i przeglądający zaczyna przepuszczać. Po każdym zadaniu jest commit, więc można cofnąć dokładnie ten krok.

Na co patrzymy w diffie, w tej kolejności:

1. Pliki spoza zadania, a zwłaszcza konfiguracja, skrypty i `.gitea/workflows`.
2. Testy: czy asercje nie zostały osłabione, testy usunięte albo oznaczone jako pomijane.
3. Nowe zależności: nazwa pakietu, autor, data powstania.
4. Wyłączone zabezpieczenia: wyłączona weryfikacja certyfikatu, szerokie uprawnienia plików, szeroki dostęp do bazy.
5. Zaszyte adresy, hasła i klucze.

> **Pojęcie: slopsquatting.** Atak, w którym ktoś rejestruje w publicznym repozytorium pakietów nazwę, którą modele często zmyślają. Badanie omawiane w 2025 roku wykazało, że w kodzie generowanym przez modele od kilku do kilkunastu procent zależności bywało nieistniejących (dla Pythona około 16 procent, dla JavaScriptu około 21), a w ponad 40 procentach przypadków ta sama zmyślona nazwa wracała przy każdym powtórzeniu. Wynik zależy od modelu i się zmienia, ale mechanizm jest stały.

W praktyce każdą nową zależność sprawdzamy ręcznie w rejestrze pakietów, zanim wejdzie do projektu.

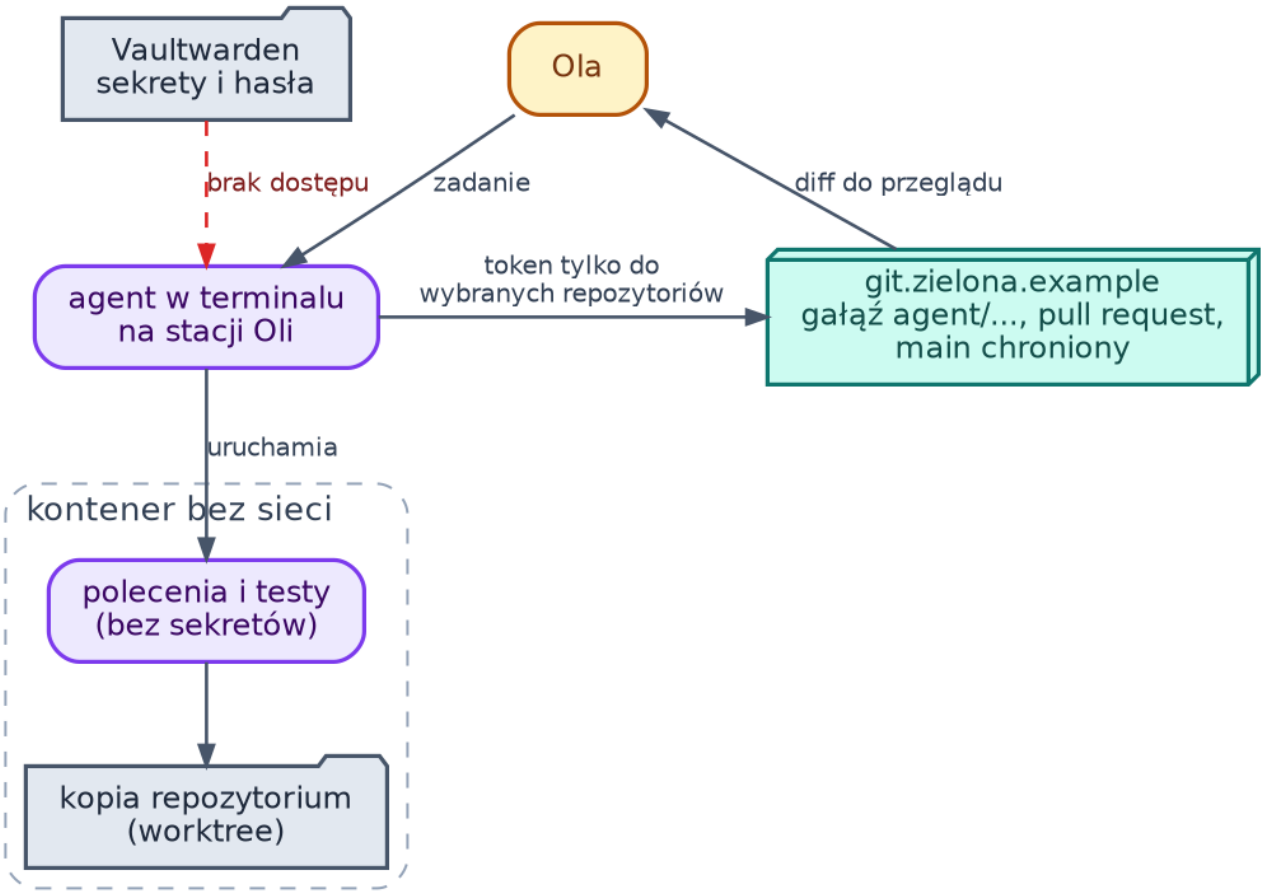
Testy

Testy dają agentowi kryterium sukcesu, a nam sposób, żeby to kryterium sprawdzić. Najlepiej działa układ, w którym najpierw piszemy test (albo opisujemy oczekiwane zachowanie), a agent dopiero potem pisze kod. Kryteria akceptacji opisuje strona Jak sprawdzić, czy to w ogóle działa.

Zielone testy nie dowodzą poprawnego kodu. Agent, który widzi czerwony test, ma dwie drogi: poprawić kod albo poprawić test, i nie zawsze wybiera pierwszą. Dlatego zmiany w testach dostają osobną uwagę przy przeglądzie, a pełny zestaw uruchamiamy w CI, na czystym środowisku, nie tylko na stacji, gdzie agent mógł coś zmienić.

Piaskownica i kontenery

> **Pojęcie: piaskownica.** Środowisko, w którym błąd albo złośliwe polecenie może zniszczyć tylko to, co do niego świadomie wstawiliśmy. W programowaniu to zwykle kontener, osobny użytkownik systemowy albo osobna maszyna wirtualna.



Agent na stacji Oli wykonuje polecenia w kontenerze bez sieci i bez sekretów, a do Gitei wysyła zmiany przez wąski token i pull request (kliknięcie otwiera pełny rozmiar)

Rozdzielamy dwie rzeczy: gdzie działa sam agent i gdzie wykonują się polecenia, które on zleca. Sam agent musi zwykle łączyć się z usługą modelu, więc ma sieć. Polecenia testów i budowania uruchamiamy w kontenerze bez sieci, z jednym katalogiem projektu i bez sekretów. Kopia projektu w osobnym worktree dodaje drugą barierę.

Metoda	Co chroni	Czego nie chroni
Kontener bez sieci z jednym katalogiem	resztę dysku, sekrety, sieć wewnętrzną	samego katalogu projektu, jeśli agent go popsuje
Osobne konto systemowe	pliki innych użytkowników	sieci i usług dostępnych z tej stacji

Metoda	Co chroni	Czego nie chroni
Osobna maszyna wirtualna	całą stację	tego, co sami do niej wpuścimy
`git worktree` na osobnej gałęzi	niezapisaną pracę człowieka	niczego poza repozytorium

Piaskownica nie zastępuje przeglądu, a przegląd nie zastępuje piaskownicy. Agent z dostępem do prywatnych danych, niezaufanej treści i sieci zewnętrznej tworzy układ opisany jako śmiertelne trio na stronie o uprawnieniach, więc jednego z tych trzech elementów odbieramy zawsze.

Sekrety w repozytorium

Sekret w repozytorium zostaje na zawsze w historii, nawet po usunięciu pliku. Jeśli trafił do commitu, uznajemy go za ujawniony: unieważniamy i wystawiamy nowy, a czyszczenie historii jest dodatkiem, nie zamiennikiem. Zasady na co dzień:

1. Sekretów nie ma w katalogu projektu, program dostaje je ze zmiennych środowiskowych albo z menedżera sekretów (u nas Vaultwarden, patrz Sekrety w Dockerze).
2. `.gitignore` wyklucza pliki z sekretami, a skan `gitleaks` działa przed commitem i w CI.
3. Agent nie dostaje dostępu do menedżera sekretów. Wartość, której potrzebuje do jednego polecenia, wstawiamy sami, na jedno uruchomienie.
4. W wynikach poleceń sekrety mogą trafić do kontekstu modelu, a stamtąd do logów dostawcy. Komendy, które wypisują zmienne środowiskowe, nie wchodzą na listę dozwolonych.

Uprawnienia agenta do Gitei i serwerów

Zasada najmniejszych uprawnień wygląda tu konkretnie. Agent ma własne konto i własny token, a nie token Oli, bo wtedy w logach nie odróżnimy jego działań od jej działań, a wyciek tokenu daje komuś wszystkie jej prawa.

Zasób	Co agent dostaje	Czego nie dostaje
Gitea	konto `agent-ola`, zapis tylko do repozytoriów zespołu, gałęzie `agent/...`	prawa administratora, prawa do organizacji, możliwość scalania do `main`
Pipeline w CI	uruchomienie testów na swojej gałęzi	sekrety wdrożeniowe, wdrożenie na produkcję
Serwery	zwykle nic, ewentualnie konto tylko do odczytu logów	klucze SSH, `sudo`, konto administratora

Zasób	Co agent dostaje	Czego nie dostaje
Baza danych	baza testowa albo konto tylko do odczytu na kopii	dostęp do produkcyjnego `pg01`
Sekrety	wartości wstawione na jedno uruchomienie	dostęp do Vaultwardena

Plik workflow w `.gitea/workflows` uruchamia się z sekretami repozytorium, więc agent, który może go zmienić i uruchomić, może te sekrety odczytać. Dlatego zmiany w workflow wymagają przeglądu człowieka, a sekrety wdrożeniowe żyją tylko w workflow dla gałęzi `main`.

Zakaz w instrukcji dla agenta nie jest zabezpieczeniem. W lipcu 2025 roku opisano głośny przypadek agenta programistycznego, który mimo wyraźnego zakazu zmian w okresie zamrożenia usunął działającą bazę danych. Tego typu granice ustawiamy uprawnieniami i ochroną gałęzi, a instrukcja jest tylko dodatkiem.

Gdy agent coś zepsuje

Najpierw odcinamy: usuwamy token agenta w Gitei albo dezaktywujemy jego konto, więc dalszych zmian nie będzie. Potem czytamy historię (`git reflog`, log Gitei) i cofamy zmiany komendą `git revert` na gałęzi, a nie przepisujemy historii. Na końcu zapisujemy, który z elementów ochrony zawiódł, i poprawiamy ustawienia, nie tylko skutki.

Częste błędy

Objaw	Przyczyna	Co robimy
W historii repozytorium jest klucz API	plik `.env` trafił do commitu, bo agent albo człowiek wykonał `git add .`	unieważniamy klucz, wystawiamy nowy, dodajemy `.env` do `.gitignore`, włączamy skan przed commitem
Testy przechodzą, a funkcja nadal działa źle	agent zmienił test zamiast kodu	czytamy diff testów osobno, ustawiamy obowiązkowy przegląd zmian w `tests/`
W `package.json` jest zależność, której nikt nie zna	model zmyślił nazwę pakietu, ktoś mógł ją zarejestrować	usuwamy, sprawdzamy w rejestrze autora i datę, zależność dodajemy ręcznie
Push na `main` przeszedł bez przeglądu	token agenta należy do administratora albo `main` nie ma ochrony	konto agenta bez praw administratora, reguła Disable Push
Kontener z agentem ma dostęp do całego hosta	zamontowane `docker.sock` albo katalog domowy	montujemy tylko katalog projektu, gniazda Dockera nigdy

Objaw	Przyczyna	Co robimy
Nie wiemy, kto zmienił plik workflow	agent pracował na koncie człowieka	osobne konto agenta i osobny token

Ćwiczenia

1. Dlaczego agent w terminalu potrzebuje innej ochrony niż podpowiedzi w edytorze?

Odpowiedź

Podpowiedź dotyczy jednej linii i człowiek akceptuje ją, zanim cokolwiek się stanie. Agent sam wykonuje polecenia i zapisuje zmiany, a jego zasięg to wszystko, do czego ma uprawnienia. Kontrola nie może więc zależeć od naszej uwagi w każdej sekundzie: ustawiamy ją z góry piaskownicą, wąskim tokenem i ochroną gałęzi.

1. Agent poprosił o dodanie zależności `dane-cennik-tools`, której Ola nie zna. Co sprawdzamy, zanim zgodzimy się na instalację?

Odpowiedź

Czy pakiet istnieje w rejestrze, kto jest autorem, kiedy powstał i ile osób go używa. Nowy pakiet o nazwie wymyślonej przez model może być specjalnie zarejestrowanym fałszywym pakietem. Jeśli nie umiemy potwierdzić pochodzenia, nie instalujemy go i wybieramy sprawdzoną bibliotekę.

1. Ola dała agentowi własny token z prawem zapisu do wszystkich repozytoriów organizacji, bo „tak jest szybciej". Wymień dwa problemy.

Odpowiedź

Po pierwsze, zasięg: agent (i każdy, kto go przejmie przez prompt injection) może zmienić każde repozytorium organizacji, nie tylko to, nad którym pracuje zespół. Po drugie, tożsamość: w logach nie odróżnimy działań agenta od działań Oli, a wyciek tokenu równa się wyciekowi całego jej konta.

1. Znajdujemy w historii repozytorium klucz do zewnętrznej usługi, dodany trzy miesiące temu i usunięty następnego dnia. Czy sprawa jest załatwiona?

Odpowiedź

Nie. Klucz został w historii i każdy, kto sklonował repozytorium, mógł go skopiować. Uznajemy go za ujawniony: unieważniamy go u dostawcy usługi i wystawiamy nowy. Dopiero potem ewentualnie czyścimy historię.

Źródła

- OWASP Top 10 for LLM Applications 2025, LLM06 Excessive Agency i LLM01 Prompt Injection: https://genai.owasp.org/llm-top-10/
- Omówienie badania o zmyślonych pakietach (slopsquatting): https://www.kaspersky.com/blog/ai-slopsquatting-supply-chain-risk/53327/
- Opis przypadku agenta, który usunął bazę danych mimo zakazu (Fortune, lipiec 2025): https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure
- Gitleaks, polecenia i konfiguracja pre-commit: https://github.com/gitleaks/gitleaks
- Gitea, zakresy tokenów dostępu: https://docs.gitea.com/development/oauth2-provider
- Docker, uruchamianie kontenerów i opcje izolacji: https://docs.docker.com/reference/cli/docker/container/run/
- Wykorzystane wcześniejsze strony książki: Uprawnienia, prompt injection i zasięg rażenia, Jak sprawdzić, czy to w ogóle działa

---

Revision #18
Created 2026-10-03 09:39:17 UTC by Tomasz Konieczny
Updated 2026-10-26 14:10:22 UTC by Tomasz Konieczny