Dlaczego zadanie nie zadziałało: diagnostyka crona i timerów

Poziom: średnio **Dotyczy: wszystkie dystrybucje (cron i systemd timers)**

Czas: ~16 min

Zadanie „powinno się uruchomić" i się nie uruchomiło — albo uruchomiło się, ale nie zrobiło tego, co miało. Ta strona to procedura krok po kroku: gdzie szukać dowodu, że w ogóle wystartowało, i jak wykluczyć po kolei najczęstsze przyczyny ciszy.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
AVC	Access Vector Cache: wpis w dzienniku SELinuksa opisujący, jaką operację zablokowano i dlaczego
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SELinux	Security-Enhanced Linux: dodatkowa warstwa uprawnień w jądrze RHEL

Fiszka

TYLKO ODCZYT

Sprawdzenie, czy demon w ogóle próbował: `journalctl -u crond --since today`

```
journalctl -u crond --since today
```

Pierwsze pytanie w diagnostyce nie brzmi „dlaczego skrypt nie zadziałał", tylko „czy cron w ogóle podjął próbę uruchomienia". Ten log odpowiada właśnie na to — jeśli nie ma tu

żadnego wpisu z nazwą zadania, problem leży w samym harmonogramie (zła składnia, zły crontab), nie w skrypcie.

Przykład:

```
wrz 18 03:00:01 app01 crond[812]: (adm_user) CMD (/usr/local/bin/backup.sh >
/var/log/backup.log 2>&1)
```

Na Debianie/Ubuntu odpowiednik to `journalctl -u cron --since today` albo `grep CRON /var/log/syslog` (patrz strona o cronie).

journalctl = journal control: czytanie dziennika systemd

TYLKO ODCZYT

Przegląd ostatnich uruchomień timera: `journalctl -u NazwaUslugi.service -n 50`

```
journalctl -u backup-nocny.service -n 50
```

Dla zadań uruchamianych przez systemd timer to jest odpowiednik powyższego — pokazuje, czy usługa wystartowała, jak się zakończyła (kod wyjścia) i co wypisała po drodze, wszystko w jednym miejscu bez szukania w plikach.

Przykład:

```
wrz 18 03:00:00 bkp01 systemd[1]: Started Kopia zapasowa katalogu danych.
wrz 18 03:00:04 bkp01 systemd[1]: backup-nocny.service: Main process exited,
code=exited, status=1/FAILURE
wrz 18 03:00:04 bkp01 systemd[1]: backup-nocny.service: Failed with result 'exit-code'.
```

Status `exit-code` z kodem różnym od zera to już dowód, że zadanie wystartowało — problem przesuwa się do samego skryptu, nie do harmonogramu.

journalctl = journal control: czytanie dziennika systemd

TYLKO ODCZYT

Sprawdzenie kolumn NEXT i LAST: `systemctl list-timers --all`

```
systemctl list-timers --all
```

`NEXT` mówi, kiedy timer uruchomi się następnym razem; `LAST` — kiedy uruchomił się ostatnio. Jeśli `LAST` jest znacznie starsze niż powinno wynikać z harmonogramu (albo pokazuje `n/a`), timer albo jest wyłączony, albo nigdy nie zadziałał.

Przykład:

```
NEXT                         LEFT      LAST                         PASSED
UNIT                ACTIVATES
n/a                          n/a       n/a                          n/a     stary-
raport.timer  stary-raport.service
Sat 2026-09-19 03:00:00 CEST 17h left  Fri 2026-09-18 03:00:00 CEST  8h ago  backup-
nocny.timer  backup-nocny.service
```

`n/a` we wszystkich kolumnach zwykle znaczy, że timer nie jest włączony (`systemctl enable --now` nigdy nie zostało wykonane) albo w ogóle nie ma jednostki `WantedBy=timers.target`.

NEXT = Near-End Crosstalk: przesłuch zbliżny, czyli ile sygnału z jednej pary przecieka do sąsiedniej pary na tym samym końcu kabla · **systemctl** = system control: sterowanie usługami systemd

TYLKO ODCZYT

Przegląd klasycznego logu tekstowego: `grep CRON /var/log/cron`

```
grep CRON /var/log/cron
```

Na systemach, gdzie `crond` loguje przez rsyslog do osobnego pliku zamiast (albo obok) journala. Szukamy wpisu z PID-em i nazwą komendy w tej samej minucie, w której zadanie miało wystartować.

Przykład:

```
Sep 18 03:00:01 app01 CRON[9821]: (adm_user) CMD (/usr/local/bin/backup.sh >
/var/log/backup.log 2>&1)
```

grep = global regular expression print: szukaj wzorca w tekście

TYLKO ODCZYT

Sprawdzenie śladu w audycie, gdy SELinux blokuje: `ausearch -m cron`

```
sudo ausearch -m cron --start today
```

Na RHEL/Rocky z aktywnym SELinux zdarza się, że cron próbuje uruchomić zadanie, ale polityka SELinux blokuje jakiś element (dostęp do pliku, wykonanie binarki spoza oczekiwanego kontekstu) — objaw wygląda jak „zadanie się nie wykonało", a przyczyna wcale nie leży w cronie ani w skrypcie. `ausearch -m cron` (albo szerzej `ausearch -m avc`) pokazuje takie odmowy.

Przykład:

```
type=AVC msg=audit(1758160801.123:456): avc:  denied  { execute } for pid=9821
comm="crond" ...
```

SELinux = Security-Enhanced Linux: dodatkowa warstwa uprawnień w jądrze RHEL · **RHEL** = Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa · **ausearch** = przeszukaj log audytu jądra, tu pod kątem zdarzeń SELinux oznaczonych avc

ZMIENIA SYSTEM

Test środowiska: `* * * * * env > /tmp/cron-env.txt`

```
echo '* * * * * env > /tmp/cron-env.txt' | crontab -
sleep 65
crontab -r
cat /tmp/cron-env.txt
```

Zamiast zgadywać, jakie zmienne widzi cron, każemy mu je po prostu wypisać do pliku. To najszybszy sposób na potwierdzenie albo wykluczenie różnic środowiska jako przyczyny.

Przykład:

```
SHELL=/bin/sh
PATH=/usr/bin:/bin
LOGNAME=adm_user
HOME=/home/adm_user
```

sleep = wstrzymaj wykonywanie skryptu na wskazaną liczbę sekund · **crontab** = cron table: tabela zadań cyklicznych · **cat** = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Porównanie `env` z crona i z powłoki

```
env > /tmp/shell-env.txt
diff /tmp/cron-env.txt /tmp/shell-env.txt
```

Różnica w `PATH` (krótszy w cronie) i brak zmiennych z `.bashrc` to niemal zawsze pierwsze, co pokaże `diff` — i niemal zawsze to jest właściwa przyczyna, gdy skrypt „działa ręcznie, a z crona nie".

Przykład:

```
< PATH=/usr/bin:/bin
< SHELL=/bin/sh
---
> PATH=/home/adm_user/.local/bin:/usr/local/bin:/usr/bin:/bin
> SHELL=/bin/bash
```

env = environment: uruchamia program z dodatkowymi zmiennymi środowiskowymi, na przykład sudo env ZMIENNA=wartość polecenie

TYLKO ODCZYT

Blokada przed nakładaniem się: `flock -n /var/lock/zadanie.lock komenda`

```
flock -n /var/lock/zadanie.lock /usr/local/bin/dlugie-zadanie.sh
```

`-n` (non-blocking) sprawia, że jeśli blokada jest już zajęta przez poprzednie, wciąż trwające uruchomienie tej samej komendy, nowa próba **natychmiast się kończy** zamiast czekać w kolejce — dokładnie to, czego chcemy przy zadaniu cyklicznym: lepiej pominąć jedno uruchomienie niż uruchomić dwie kopie naraz.

Przykład:

```
$ flock -n /var/lock/zadanie.lock /usr/local/bin/dlugie-zadanie.sh; echo $?
1
```

Kod wyjścia `1` (przy `-n`) oznacza „blokada zajęta, nie uruchomiono" — cicha porażka, nie awaria zadania.

flock = zablokuj plik na czas trwania polecenia, żeby dwa uruchomienia skryptu się nie nałożyły

🔒 TYLKO ODCZYT

Sucha weryfikacja katalogu cron.*: `run-parts --test`

```
run-parts --test /etc/cron.daily
```

Pokazujemy, które pliki z katalogu `run-parts` faktycznie by uruchomił, bez ich wykonywania — przydatne, gdy podejrzewamy, że własny skrypt wrzucony do `/etc/cron.daily/` jest pomijany, bo brakuje mu prawa wykonywania albo ma niewłaściwą nazwę (kropkę, rozszerzenie zastrzeżone przez menedżera pakietów).

Przykład:

```
/etc/cron.daily/logrotate
/etc/cron.daily/man-db.cron
```

Brak własnego skryptu na tej liście, mimo że plik istnieje w katalogu, to natychmiastowy sygnał: sprawdzamy uprawnienia (`chmod +x`) i nazwę pliku.

🔒 TYLKO ODCZYT

Weryfikacja wyrażenia OnCalendar: `systemd-analyze calendar`

```
systemd-analyze calendar "Mon *-*-* 06:00" --iterations=3
```

Ten sam sposób sprawdzenia co przy tworzeniu timera (strona o systemd timers), tu w roli diagnostycznej: jeśli timer uruchamia się „w złym dniu" albo „o złej godzinie", to pierwsze narzędzie, które potwierdza albo obala, czy problem leży w samym wyrażeniu.

Przykład:

```
Next elapse: Mon 2026-09-21 06:00:00 CEST
(iter. 2): Mon 2026-09-28 06:00:00 CEST
(iter. 3): Mon 2026-10-05 06:00:00 CEST
```

🔒 TYLKO ODCZYT

Zachowanie ciszy przy sukcesie: `chronic` z moreutils

```
chronic /usr/local/bin/backup.sh
```

`chronic` (pakiet `moreutils`) uruchamia komendę i **połyka** całe jej wyjście, jeśli zakończyła się kodem 0 (sukces) — wypisuje je dopiero, jeśli komenda zawiedzie. To odwraca domyślne zachowanie cron: zamiast dostawać mail za każdym razem (patrz strona o cronie), dostajemy go wyłącznie wtedy, gdy naprawdę coś poszło nie tak.

Przykład:

```
0 2 * * * chronic /usr/local/bin/backup.sh
```

◫◫ ZMIENIA SYSTEM

Przekierowanie do logu z datą w nazwie

```
/usr/local/bin/backup.sh >> /var/log/backup-$(date +\%Y\%m\%d).log 2>&1
```

Każde uruchomienie trafia do osobnego pliku dziennego — ułatwia porównanie „co się stało wczoraj kontra dziś" bez przeglądania jednego rosnącego pliku. Pamiętamy o escapowaniu `%` w crontabie (strona o cronie).

◫◫ TYLKO ODCZYT

Zabezpieczenie skryptu przed cichym błędem: `set -euo pipefail`

```
#!/bin/bash
set -euo pipefail

/usr/local/bin/krok1.sh
/usr/local/bin/krok2.sh
```

`set -e` przerywa skrypt na pierwszej komendzie zakończonej błędem zamiast kontynuować z fałszywym poczuciem sukcesu; `-u` zamienia użycie niezdefiniowanej zmiennej w błąd zamiast cichego podstawienia pustego ciągu; `-o pipefail` sprawia, że potok `a | b` zwraca błąd, jeśli zawiedzie `a`, nawet gdy `b` się powiedzie. Bez tych trzech opcji skrypt uruchomiony ręcznie i przerwany przez Ctrl+C na widocznym błędzie z crona po prostu leci dalej i kończy się kodem 0, mimo że w środku coś zawiodło.

◫◫ TYLKO ODCZYT

Wszystkie narzędzia diagnostyczne — ściąga

Narzędzie	Do czego
`journalctl -u crond` / `nazwa.service`	czy zadanie w ogóle wystartowało
`systemctl list-timers --all`	kiedy ostatnio/następnym razem, czy timer włączony
`grep CRON /var/log/cron`	log tekstowy tam, gdzie nie ma journala
`ausearch -m cron`	czy SELinux zablokował coś po drodze
`env > plik` w crontabie	realne środowisko widziane przez zadanie
`flock -n`	zapobieganie nakładaniu się uruchomień
`run-parts --test`	czy skrypt w katalogu cron.* w ogóle zostanie zauważony
`systemd-analyze calendar`	czy wyrażenie OnCalendar znaczy to, co myślisz
`chronic`	log tylko przy błędzie, cisza przy sukcesie

Uważaj

- Brak wpisu w journalu/logu crona to zupełnie inny problem niż wpis z błędem — nie mylić „nie wystartowało" z „wystartowało i zawiodło".
- `flock -n` bez `-n` **czeka** na zwolnienie blokady zamiast pomijać uruchomienie — łatwo pomylić zachowanie obu wariantów.
- `ausearch` wymaga `auditd` uruchomionego i włączonego logowania zdarzeń AVC — na maszynie z wyłączonym audytem nic nie znajdzie, mimo że SELinux faktycznie blokuje.

SELinux = Security-Enhanced Linux: dodatkowa warstwa uprawnień w jądrze RHEL · **AVC** = Access Vector Cache: wpis w dzienniku SELinuksa opisujący, jaką operację zablokowano i dlaczego

Z życia wzięte

Z ŻYCIA WZIĘTE

Czwartek, 3:05: backup na app01 kończy się cicho kodem 0, ale plik docelowy w `/var/backups/` ma wczorajszą datę. `set -e` nie było na górze skryptu, więc nieudane `cd /mnt/archiwum` w środku nie przerwało reszty, a ostatnia linia i tak zwróciła sukces. Dopiero uruchomienie tej samej komendy w odtworzonym, ubogim środowisku crona (`env -i PATH=/usr/bin:/bin`) pokazuje błąd, który w logu crona był niewidoczny.

Po co to istnieje

> **Pojęcie: środowisko procesu potomnego.** Każdy proces uruchomiony przez inny proces (proces potomny) dziedziczy po nim pewien zestaw danych startowych: zmienne środowiskowe, katalog roboczy, maskę uprawnień (`umask`), a czasem też otwarte deskryptory plików. To, co dokładnie proces potomny dziedziczy, zależy od tego, kto go uruchamia — zadanie startowane ręcznie z interaktywnej powłoki dziedziczy bogate środowisko tej powłoki; zadanie startowane przez `cron` albo `systemd` dziedziczy tylko to, co te programy same przygotowały, zwykle znacznie skromniejszy zestaw. Skrypty administracyjne pisze się i testuje niemal zawsze w tym pierwszym, bogatym środowisku — a uruchamia w tym drugim, ubogim — i cała ta strona jest w gruncie rzeczy o tym jednym rozjeździe oraz o narzędziach, które pozwalają go zmierzyć zamiast zgadywać.

Diagnostyka zadań cyklicznych ma tę specyfikę, że nie da się „stanąć obok" i patrzeć, jak coś zawodzi na żywo — zdarzenie już minęło, zanim ktoś zauważył brak efektu. Dlatego cała procedura opiera się na artefaktach zostawionych po fakcie: logach, plikach stanu, kodach wyjścia — i na eksperymentach odtwarzających warunki startowe zadania, żeby zobaczyć to, czego nie było widać podczas oryginalnego, nieudanego uruchomienia.

Analogia

Diagnozowanie martwego zadania cyklicznego przypomina **pracę śledczego po fakcie, nie strażaka gaszącego pożar na żywo**.

Strażak przyjeżdża w trakcie zdarzenia i widzi płomienie bezpośrednio. Śledczy przyjeżdża następnego dnia do zgliszcz i musi zrekonstruować, co się stało, wyłącznie z tego, co zostało: śladów (logi), zeznań świadków (kody wyjścia, `systemctl status`), eksperymentów odtwarzających warunki (u nas: to samo środowisko, ta sama komenda). Śledczy, który od razu zakłada winnego bez sprawdzenia śladów, zwykle się myli — dokładnie tak samo administrator, który obwinia „coś z uprawnieniami" bez sprawdzenia logu, traci czas na naprawianie rzeczy, które wcale nie były zepsute.

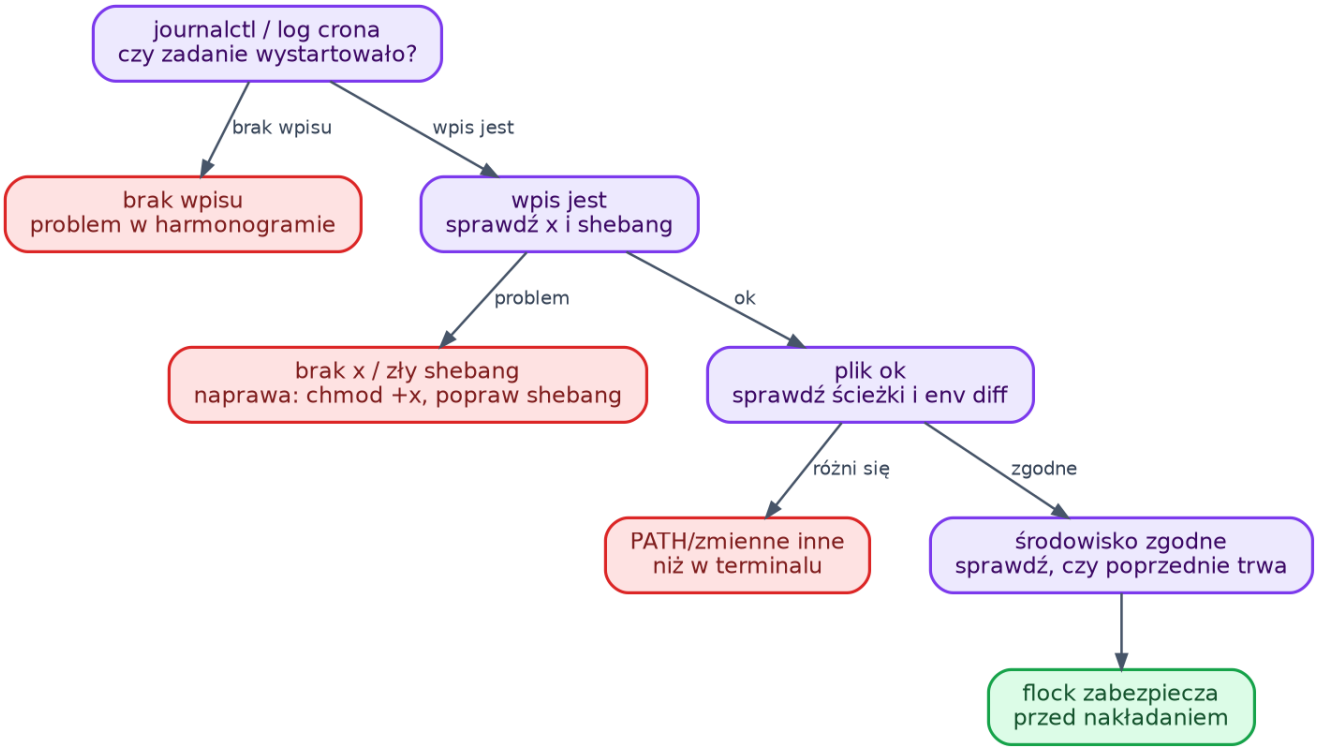
Jak to działa

Rozwiń: jak to działa pod spodem

> **Pojęcie: blokada współbieżności.** Mechanizm zapewniający, że tylko jedna instancja danego zadania działa naraz — realizowany zwykle przez plik blokady (`/var/lock/nazwa.lock`), który pierwszy uruchomiony proces zakłada, a kolejne próby uruchomienia sprawdzają przed startem właściwej pracy. `flock` to gotowe narzędzie do tego celu, oparte na blokadach plikowych jądra (`flock(2)`): trzyma otwarty deskryptor pliku blokady przez cały czas trwania komendy, a system sam usuwa blokadę, jeśli proces się zakończy (nawet awaryjnie) — co eliminuje klasyczny problem „osierocony plik blokady" znany z ręcznie pisanych skryptów sprawdzających istnienie pliku `.lock` .

Zadania cykliczne są szczególnie narażone na nakładanie się, bo harmonogram nie sprawdza, czy poprzednie uruchomienie się zakończyło — po prostu odpala kolejne o wyznaczonej porze. Zadanie, które zwykle trwa 30 sekund, ale raz — z powodu przeciążonego dysku — trwa 90 sekund, przy interwale co minutę nałoży się samo na siebie. Dwie kopie walczące o te same zasoby (blokady w bazie, ten sam plik wyjściowy) to częsta przyczyna sporadycznych awarii, które znikają, gdy administrator próbuje uruchomić zadanie ręcznie (bo wtedy uruchamia tylko jedną kopię).

Procedura diagnostyczna krok po kroku.



Procedura diagnostyczna: od potwierdzenia startu zadania, przez uprawnienia i środowisko, po sprawdzenie współbieżności (kliknięcie otwiera pełny rozmiar)

1. **Czy zadanie w ogóle się uruchomiło?** `journalctl -u crond` / `nazwa.service` albo `grep CRON /var/log/cron`. Brak wpisu = problem w harmonogramie (zła składnia crontaba, timer nieaktywny, `cron.allow` / `at.allow` blokuje użytkownika), nie w skrypcie.
2. **Czy skrypt jest wykonywalny?** `ls -l skrypt.sh` — brak `x` w uprawnieniach kończy się cichym pominięciem w katalogach `cron.daily` / `run-parts`, a jawnym `Permission denied` przy bezpośrednim wywołaniu z crontaba.
3. **Czy ma poprawny shebang?** Pierwsza linia (`#!/bin/bash`, `#!/usr/bin/env python3`) musi wskazywać istniejący interpreter pod poprawną ścieżką — inaczej `exec format error` albo uruchomienie przez zły interpreter (np. `/bin/sh` zamiast `/bin/bash`, co bywa krytyczne przy składni specyficznej dla basha).
4. **Czy ścieżki wewnątrz skryptu są bezwzględne?** Skrypt odwołujący się do plików przez ścieżki względne zależy od katalogu roboczego procesu — a katalog roboczy crona bywa inny niż ten, w którym testowałeś skrypt ręcznie.
5. **Czy środowisko wystarcza?** Test przez `env > plik` w crontabie i `diff` z powłoką interaktywną (Fiszka wyżej) — to pytanie odpowiada samo za siebie, jeśli masz oba pliki obok siebie.
6. **Czy poprzednie uruchomienie jeszcze trwa?** `ps aux | grep skrypt.sh` w trakcie podejrzanego okresu, albo trwałe rozwiązanie: `flock` na stałe wpięty w komendę, żeby to pytanie nigdy więcej nie było zgadywanką.

Ta kolejność nie jest przypadkowa — każdy kolejny krok zakłada, że poprzedni został wykluczony, więc przeskakiwanie od razu do punktu 5 czy 6 bez sprawdzenia 1–4 zwykle kończy się szukaniem złożonego wyjaśnienia dla prostego problemu (brakującego `chmod +x`).

Krok po kroku

1. Odtwórz podejrzenie: zadanie „nic nie robi".

```
cat /var/log/backup.log
```

```
(pusty plik)
```

2. Sprawdź, czy w ogóle wystartowało.

```
journalctl -u crond --since "1 hour ago" | grep backup
```

```
wrz 18 03:00:01 app01 crond[812]: (adm_user) CMD (/usr/local/bin/backup.sh >
/var/log/backup.log 2>&1)
```

Wystartowało — problem przesuwa się do skryptu.

3. Sprawdź uprawnienia i shebang.

```
ls -l /usr/local/bin/backup.sh
head -1 /usr/local/bin/backup.sh
```

```
-rwxr-xr-x 1 root root 412 wrz 10 10:00 /usr/local/bin/backup.sh
#!/bin/bash
```

Wykonywalny, shebang poprawny — dalej.

4. Odtwórz środowisko crona i uruchom ręcznie w nim.

```
env -i PATH=/usr/bin:/bin SHELL=/bin/sh /usr/local/bin/backup.sh
```

```
/usr/local/bin/backup.sh: line 4: aws: command not found
```

Znaleziony winowajca: skrypt wywołuje `aws` bez ścieżki bezwzględnej, a `env -i` z ubogim `PATH` (dokładnie takim, jaki ma cron) nie widzi katalogu, w którym `aws` jest zainstalowane.

5. Napraw i zweryfikuj ponownie tą samą metodą.

```
sed -i 's|aws |/usr/local/bin/aws |' /usr/local/bin/backup.sh
env -i PATH=/usr/bin:/bin SHELL=/bin/sh /usr/local/bin/backup.sh
```

Częste błędy

Zadanie „nic nie robi", a log crona pokazuje, że wystartowało z kodem 0.

```
crond[812]: (adm_user) CMD (/usr/local/bin/backup.sh)
```

Kod wyjścia 0 nie znaczy „skrypt zrobił to, co miał", tylko „ostatnia komenda w skrypcie zakończyła się sukcesem". Skrypt bez `set -e` może zawieść w środku (np. `cd /brakujacy/katalog` się nie uda) i mimo to dociągnąć do końca z sukcesem na ostatniej linii. Naprawa: `set -euo pipefail` na górze każdego skryptu uruchamianego z crona.

Dwie kopie tego samego zadania walczą o ten sam plik wyjściowy.

```
tar: file changed as we read it
```

Poprzednie uruchomienie długiego backupu jeszcze trwa, gdy harmonogram odpala kolejne — oba procesy jednocześnie czytają/piszą te same pliki. Objaw bywa sporadyczny i trudny do powiązania

z przyczyną, bo nie zawsze się nakładają. Naprawa: `flock -n /var/lock/backup.lock /usr/local/bin/backup.sh` wpięte na stałe do crontaba.

SELinux blokuje działanie, a komunikat w logu crona nic o tym nie mówi.

```
crond[812]: (adm_user) CMD (/usr/local/bin/skaner.sh)
```

Log crona potwierdza start, sam skrypt milczy albo kończy się niejasnym błędem, a przyczyna leży w odmowie SELinux, o której cron w ogóle nie wie (bo to jądro blokuje operację, nie cron). Naprawa: `ausearch -m avc --start today` albo `sealert -a /var/log/audit/audit.log`, jeśli pakiet `setroubleshoot` jest zainstalowany.

`run-parts` pomija skrypt bez żadnego komunikatu.

```
run-parts --test /etc/cron.daily
```

```
/etc/cron.daily/logrotate
```

Własny skrypt `moj-skrypt.sh` wrzucony do `/etc/cron.daily/` nie pojawia się na liście — `run-parts` po cichu pomija pliki z kropką w nazwie (traktowane jako rozszerzenie, np. `.sh`, `.bak`) na niektórych konfiguracjach, oraz pliki bez prawa wykonywania, bez żadnego ostrzeżenia. Naprawa: `chmod +x`, a nazwę pliku bez kropki (`moj-skrypt`, nie `moj-skrypt.sh`) — zależnie od dystrybucji i wersji `run-parts` reguła bywa różna, więc `--test` zawsze rozstrzyga wątpliwość.

Timer wygląda na aktywny, ale `LAST` nigdy się nie zmienia.

```
NEXT                         LAST  UNIT
Fri 2026-09-19 03:00:00 CEST n/a   raport.timer
```

Timer jest włączony i ma wyznaczony następny termin, ale kolumna `LAST` pokazuje `n/a` mimo że powinien już się uruchomić kilka razy. To zwykle znaczy, że jednostka `.service` powiązana z timerem nie istnieje albo ma literówkę w nazwie — `systemctl cat raport.service` potwierdzi, czy plik w ogóle jest tam, gdzie systemd go szuka.

□□ Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora poza jednym zaznaczonym miejscem.

Ćwiczenie 1 — odtworzenie ubogiego środowiska bez czekania na crona

```
cd /tmp
cat > skrypt-testowy.sh <<'EOF'
#!/bin/bash
set -euo pipefail
echo "Start: $(date)"
which terraform || echo "terraform nie znaleziony w PATH"
EOF
chmod +x skrypt-testowy.sh

./skrypt-testowy.sh
echo "--- teraz w środowisku crona ---"
env -i PATH=/usr/bin:/bin SHELL=/bin/sh ./skrypt-testowy.sh
```

Dlaczego wynik drugiego uruchomienia różni się od pierwszego, mimo że to dokładnie ten sam plik?

Odpowiedź

`env -i` czyści całe środowisko i buduje je od nowa tylko z podanych zmiennych — dokładnie tak ubogie, jak środowisko cron opisane na stronie o cronie. Jeśli `terraform` jest zainstalowany w katalogu spoza `/usr/bin:/bin` (np. `/usr/local/bin` albo katalog domowy), pierwsze uruchomienie (z pełnym środowiskiem powłoki) go znajdzie, a drugie — nie. To jest właśnie test, który wykrywa problemy z `PATH` bez konieczności czekania na faktyczne uruchomienie przez crona.

Ćwiczenie 2 — `flock` zapobiega nakładaniu się

```
cd /tmp
cat > dlugie-zadanie.sh <<'EOF'
#!/bin/bash
echo "$(date +%T) start PID $$" >> /tmp/flock-test.log
sleep 5
```

```
echo "$(date +%T) koniec PID $$" >> /tmp/flock-test.log
EOF
chmod +x dlugie-zadanie.sh
rm -f /tmp/flock-test.log

flock -n /tmp/zadanie.lock ./dlugie-zadanie.sh &
sleep 1
flock -n /tmp/zadanie.lock ./dlugie-zadanie.sh; echo "kod wyjścia drugiej próby: $?"
wait
cat /tmp/flock-test.log
```

Ile razy skrypt faktycznie się wykonał (start+koniec w logu), mimo że próbowano go uruchomić dwa razy? Co pokazał kod wyjścia drugiej próby?

Odpowiedź

Skrypt wykonuje się tylko raz — druga próba, uruchomiona sekundę po pierwszej, zastaje blokadę zajętą i `flock -n` natychmiast rezygnuje, zwracając kod wyjścia `1`, bez uruchamiania `dlugie-zadanie.sh` w ogóle. W logu widać tylko jedną parę „start/koniec". Bez `flock` obie próby uruchomiłyby się równolegle, dopisując przeplecione wpisy do tego samego pliku.

Pytania kontrolne

1. Masz do wyboru dwa fakty: „log crona nie zawiera żadnego wpisu o zadaniu" i „log crona pokazuje uruchomienie, ale skrypt nic nie zrobił". Dlaczego ważne jest rozróżnienie tych dwóch sytuacji na samym początku diagnozy?

Odpowiedź

Bo wskazują na zupełnie inne miejsce problemu. Brak wpisu w logu crona oznacza, że problem jest w samym harmonogramie — zła składnia crontaba, wyłączony timer, blokada przez `cron.allow`/`at.allow` — i szukanie błędu w skrypcie jest wtedy stratą czasu. Wpis potwierdzający start przesuwa cały ciężar diagnozy do wnętrza skryptu: uprawnień, shebangu, środowiska, ścieżek.

2. Dlaczego skrypt może zakończyć się kodem wyjścia 0, mimo że w środku wystąpił poważny błąd?

Odpowiedź

Bo bez `set -e` powłoka domyślnie kontynuuje wykonywanie kolejnych linii nawet po nieudanej komendzie, a kod wyjścia całego skryptu to kod wyjścia jego **ostatniej** komendy — jeśli ta akurat się powiedzie, cały skrypt zgłasza sukces niezależnie od tego, co zawiodło wcześniej. `set -euo pipefail` na górze skryptu eliminuje ten problem.

3. Timer systemd ma `Active: active (waiting)` i poprawny `Trigger:` w przyszłości, ale zadanie nigdy realnie nie robi tego, co powinno. Jaki jest następny krok diagnozy?

Odpowiedź

Sprawdzić `journalctl -u nazwa.service` dla powiązanej usługi — `waiting` na timerze potwierdza tylko, że harmonogram działa poprawnie, nic nie mówi o tym, co dzieje się wewnątrz uruchamianej usługi. Kod wyjścia różny od zera i treść logu usługi pokażą, czy to problem środowiska, uprawnień, czy logiki samego skryptu.

4. Zadanie trwa czasem dłużej niż zwykle (np. przy wolniejszym dysku) i interwał uruchomień bywa krótszy niż typowy czas trwania. Jakie jedno narzędzie zabezpiecza przed najpoważniejszym skutkiem tej sytuacji i jak dokładnie działa?

Odpowiedź

`flock`, najczęściej w wariancie `flock -n plik.lock komenda`. Zakłada blokadę plikową na czas trwania komendy; jeśli kolejne uruchomienie zastaje blokadę zajętą, `-n` (non-blocking) każe mu natychmiast zrezygnować zamiast czekać albo uruchamiać drugą kopię — zapobiega to nałożeniu się dwóch instancji tego samego zadania na tych samych zasobach.

Powiązane: cron i crontab: zadania cykliczne · systemd timers: nowocześniejsza alternatywa dla crona · Zadania jednorazowe: at i batch · journalctl: przeszukiwanie dziennika systemd

Źródła: `man 1 flock` · `man 8 run-parts` · `man 1 systemd-analyze` · `man 8 ausearch` · `man 1 chronic` (moreutils) · `man 1 journalctl`

Revision #15
Created 2026-09-18 06:57:11 UTC by Tomasz Konieczny
Updated 2026-10-03 19:09:49 UTC by Tomasz Konieczny