# SSH: klucze, ssh-agent, config, hartowanie

**Poziom: średni**

**Dotyczy: RHEL/Rocky 9–10 (OpenSSH), różnice dla Debian/Ubuntu i SLES zaznaczone**

**Czas: ~18 min**

*Logowanie kluczem SSH zastępuje hasło parą matematycznie powiązanych plików: jeden zostaje na serwerach, drugi nigdy nie opuszcza Twojej maszyny. Uprawnienia na tych plikach nie są formalnością — `sshd` odmówi klucza, którego nie da się chronić przed innymi, zanim w ogóle sprawdzi, czy pasuje.*

**Skróty użyte na tej stronie (7)**

| Skrót | Co znaczy |
|---|---|
| **MAC** | Media Access Control: unikalny adres fizyczny karty sieciowej, przypisany fabrycznie |
| **RHEL** | Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa |
| **RSA** | Rivest, Shamir, Adleman: algorytm szyfrowania kluczem publicznym, którym Vaultwarden podpisuje tokeny sesji, stąd pliki rsa_key* w danych aplikacji |
| **SFTP** | SSH File Transfer Protocol: bezpieczne przesyłanie plików przez SSH |
| **SLES** | SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL |
| **SSH** | Secure Shell: szyfrowane zdalne logowanie do serwera |
| **TLS** | Transport Layer Security: szyfrowanie połączeń, np. w HTTPS |

# Fiszka

**TYLKO ODCZYT**

## Wygenerowanie klucza (algorytm zalecany): ssh-keygen -t ed25519

```
ssh-keygen -t ed25519
```

Tworzymy nową parę kluczy opartą na krzywych eliptycznych — krótszy klucz niż RSA przy równoważnym poziomie bezpieczeństwa, szybsza weryfikacja podpisu, domyślny wybór na współczesnych systemach. Komenda pyta o lokalizację pliku (domyślnie `~/.ssh/id_ed25519`) i opcjonalne hasło zabezpieczające sam klucz (passphrase).

Przykład:

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/adm_user/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/adm_user/.ssh/id_ed25519
Your public key has been saved in /home/adm_user/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:k3Jz9QpXv2mN8dL4wRt7YbHc1FgKs6TnWaEo0IqU5xA adm_user@srv01
```

**RSA** = Rivest, Shamir, Adleman: algorytm szyfrowania kluczem publicznym, którym Vaultwarden podpisuje tokeny sesji, stąd pliki rsa_key* w danych aplikacji · **ssh-keygen** = wygeneruj parę kluczy SSH albo sprawdź odcisk klucza

**TYLKO ODCZYT**

## Wygenerowanie klucza (wariant zapasowy): ssh-keygen -t rsa -b 4096

```
ssh-keygen -t rsa -b 4096
```

RSA to starsza rodzina algorytmów oparta na faktoryzacji dużych liczb pierwszych — wybieramy ją, gdy docelowy system jest na tyle stary, że nie obsługuje kryptografii krzywych eliptycznych. `-b 4096` ustawia długość klucza w bitach; 2048 to obecne minimum uznawane za bezpieczne, 4096 to bezpieczny margines na kolejne lata, kosztem odrobinę wolniejszych operacji.

**RSA** = Rivest, Shamir, Adleman: algorytm szyfrowania kluczem publicznym, którym Vaultwarden podpisuje tokeny sesji, stąd pliki rsa_key* w danych aplikacji · **ssh-keygen** = wygeneruj parę kluczy SSH albo sprawdź odcisk klucza

**TYLKO ODCZYT**

## Odczyt odcisku istniejącego klucza: ssh-keygen -lf

```
ssh-keygen -lf ~/.ssh/id_ed25519.pub
```

Wyliczamy odcisk (fingerprint) podanego pliku klucza, bez generowania niczego nowego. Ten sam odcisk powinien pojawić się po obu stronach połączenia — u nas i w logach serwera — gdy porównujemy, czy klucz, którym się logujemy, to na pewno ten, o który chodzi.

Przykład:

```
256 SHA256:k3Jz9QpXv2mN8dL4wRt7YbHc1FgKs6TnWaEo0IqU5xA adm_user@srv01 (ED25519)
```

**ssh-keygen** = wygeneruj parę kluczy SSH albo sprawdź odcisk klucza

**TYLKO ODCZYT**

## Instalacja klucza publicznego na zdalnym serwerze: ssh-copy-id

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub nazwa_usera@NazwaSerwera
```

Łączymy się ze wskazanym serwerem (hasłem, bo klucza tam jeszcze nie ma), dopisujemy treść wskazanego klucza publicznego do `~/.ssh/authorized_keys` docelowego użytkownika — komenda po drodze sama ustawia poprawne uprawnienia katalogu `.ssh` i pliku `authorized_keys` — dokładnie te, których brak opisuje sekcja *Częste błędy*.

Przykład:

```
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/home/adm_user/.ssh/id_ed25519.pub"
Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'user1@srv02'"
and check to make sure that only the key(s) you wanted were added.
```

**ssh-copy-id** = skopiuj własny klucz publiczny SSH na serwer i dopisz go do authorized_keys, dostępne na Linuksie i macOS, brak w Windows

**ZMIENIA SYSTEM**

## Ręczne dopisanie klucza do `authorized_keys`

```
cat ~/.ssh/id_ed25519.pub | ssh nazwa_usera@NazwaSerwera 'mkdir -p ~/.ssh && cat >>
~/.ssh/authorized_keys && chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys'
```

Robimy ręcznie dokładnie to, co robi `ssh-copy-id` — przydatne, gdy `ssh-copy-id` nie jest dostępne (np. minimalny obraz kontenera) albo gdy klucz trzeba wgrać przez zupełnie inny kanał (konsola, system zarządzania konfiguracją), a nie przez istniejące połączenie SSH.

**SSH** = Secure Shell: szyfrowane zdalne logowanie do serwera · **cat** = concatenate: sklej i wypisz pliki

**TYLKO ODCZYT**

## Uruchomienie agenta trzymającego odszyfrowane klucze w pamięci: ssh-agent

```
eval "$(ssh-agent -s)"
```

Uruchamiamy w tle proces przechowujący nasze odszyfrowane klucze prywatne przez czas trwania sesji — dzięki temu hasło zabezpieczające klucz (passphrase) wpisujemy raz, a nie przy każdym kolejnym połączeniu SSH. `eval` ustawia w bieżącej powłoce zmienne środowiskowe (`SSH_AUTH_SOCK`, `SSH_AGENT_PID`), którymi klienci SSH odnajdują działający agent.

Przykład:

```
Agent pid 4821
```

**SSH** = Secure Shell: szyfrowane zdalne logowanie do serwera

**TYLKO ODCZYT**

## Dodanie klucza do działającego agenta: ssh-add

```
ssh-add ~/.ssh/id_ed25519
```

Odszyfrowujemy wskazany klucz (pytając o passphrase, jeśli klucz ją ma) i przekazujemy go uruchomionemu wcześniej `ssh-agent`. Bez tego kroku sam uruchomiony agent jest pusty i niczego nie przyspiesza.

Przykład:

```
Enter passphrase for /home/adm_user/.ssh/id_ed25519:
Identity added: /home/adm_user/.ssh/id_ed25519 (adm_user@srv01)
```

**□□ TYLKO ODCZYT**

## Wyświetlenie listy kluczy załadowanych do agenta: ssh-add -l

```
ssh-add -l
```

Pokazujemy odciski wszystkich kluczy, które agent obecnie trzyma w pamięci — bez tego nie sposób odróżnić „agent działa, ale jest pusty” od „agent ma załadowany zły klucz”.

Przykład:

```
256 SHA256:k3Jz9QpXv2mN8dL4wRt7YbHc1FgKs6TnWaEo0IqU5xA adm_user@srv01 (ED25519)
```

Jeśli agent nie ma żadnego klucza, ta sama komenda odpowie `The agent has no identities.`

**□□ TYLKO ODCZYT**

## Logowanie wskazanym kluczem z pominięciem agenta i configu: ssh -i

```
ssh -i ~/.ssh/id_klucz_specjalny nazwa_usera@NazwaSerwera
```

Wymuszamy użycie konkretnego pliku klucza dla tego jednego połączenia, niezależnie od tego, co jest załadowane w agencie i co mówi `~/.ssh/config`. Przydatne przy kluczach jednorazowych albo gdy jedno konto ma kilka kluczy do różnych celów.

□□ **ssh** = Secure Shell: szyfrowane zdalne logowanie

**□□ TYLKO ODCZYT**

## Zdefiniowanie profilu połączenia: ~/.ssh/config

```
Host produkcja
    HostName srv01.ad.zielona.example
    User adm_user
    IdentityFile ~/.ssh/id_produkcja
    Port 2222

Host baza-danych
    HostName 10.20.10.33
```

```
    User postgres
    IdentityFile ~/.ssh/id_bazy
```

Zamiast pamiętać pełną komendę z hostem, portem, użytkownikiem i kluczem, definiujemy skrót raz i łączymy się przez `ssh produkcja`. Klienty inne niż `ssh` (np. `scp`, `rsync -e ssh`) też respektują ten plik.

```
ssh produkcja
```

**ssh** = Secure Shell: szyfrowane zdalne logowanie

**TYLKO ODCZYT**

## Przeskok przez host pośredniczący (jump host): ssh -J

```
ssh -J bastion.ad.zielona.example nazwa_usera@10.20.10.33
```

Łączymy się najpierw z hostem pośredniczącym (`bastion.ad.zielona.example`), a stamtąd, w ramach tego samego, jednego wywołania klienta, otwieramy kolejne połączenie do docelowego hosta wewnętrznego, który nie ma bezpośredniego dostępu z zewnątrz. Odpowiednik starszego, dwuetapowego wzorca `ProxyCommand`, ale bez potrzeby ręcznego wywoływania `nc`/`ssh -W` na hoście pośredniczącym.

**ssh** = Secure Shell: szyfrowane zdalne logowanie

**TYLKO ODCZYT**

## Wyświetlenie listy obsługiwanych algorytmów MAC: ssh -Q mac

```
ssh -Q mac
```

Wypisujemy pełną listę algorytmów integralności wiadomości (MAC), które **klient** SSH potrafi zaproponować podczas negocjacji połączenia — nie to, co faktycznie zostanie użyte, tylko co jest w ogóle dostępne w tej wersji OpenSSH.

Przykład:

```
hmac-sha1
hmac-sha1-96
hmac-sha2-256
hmac-sha2-512
```

```
hmac-md5
hmac-md5-96
hmac-ripemd160
umac-64@openssh.com
umac-128@openssh.com
hmac-sha1-etm@openssh.com
hmac-sha2-256-etm@openssh.com
hmac-sha2-512-etm@openssh.com
hmac-md5-etm@openssh.com
hmac-md5-96-etm@openssh.com
hmac-ripemd160-etm@openssh.com
umac-64-etm@openssh.com
umac-128-etm@openssh.com
```

□□ **MAC** = Media Access Control: unikalny adres fizyczny karty sieciowej, przypisany fabrycznie · **SSH** = Secure Shell: szyfrowane zdalne logowanie do serwera · **ssh** = Secure Shell: szyfrowane zdalne logowanie

□□ TYLKO ODCZYT

## Wyświetlenie listy obsługiwanych algorytmów wymiany kluczy: ssh -Q kex

```
ssh -Q kex
```

To samo pytanie, tylko o algorytmy wymiany kluczy (key exchange) — mechanizm, którym obie strony uzgadniają wspólny sekret sesji, zanim jeszcze cokolwiek zostanie zaszyfrowane.

Przykład:

```
curve25519-sha256
curve25519-sha256@libssh.org
diffie-hellman-group16-sha512
diffie-hellman-group18-sha512
diffie-hellman-group14-sha256
diffie-hellman-group-exchange-sha256
```

□□ **ssh** = Secure Shell: szyfrowane zdalne logowanie

□□ TYLKO ODCZYT

## Wyświetlenie listy obsługiwanych szyfrów: ssh -Q cipher

```
ssh -Q cipher
```

Analogicznie dla szyfrów symetrycznych, którymi faktycznie szyfrowana jest treść sesji po ustaleniu klucza.

Przykład:

```
chacha20-poly1305@openssh.com
aes128-ctr
aes192-ctr
aes256-ctr
aes128-gcm@openssh.com
aes256-gcm@openssh.com
```

**ssh** = Secure Shell: szyfrowane zdalne logowanie

**TYLKO ODCZYT**

## Sprawdzenie, co serwer faktycznie oferuje: sshd -T

```
sudo sshd -T | grep -E '^(macs|ciphers|kexalgorithms)'
```

`ssh -Q` mówi, co klient *potrafi*; `sshd -T` mówi, co konkretny serwer, po uwzględnieniu całego `/etc/ssh/sshd_config` (razem z blokami `Match`), *faktycznie oferuje* w negocjacji. To dwie różne odpowiedzi na dwa różne pytania, i diagnoza problemów z negocjacją algorytmów wymaga sprawdzenia obu stron.

Przykład:

```
kexalgorithms curve25519-sha256,curve25519-sha256@libssh.org,diffie-hellman-group16-
sha512
ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com,aes256-ctr
macs umac-128-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-sha2-512-
etm@openssh.com
```

Widać tu typowy efekt hartowania: administrator ograniczył serwer do samych wariantów `-etm@openssh.com` (encrypt-then-MAC — nowocześniejszy, bezpieczniejszy porządek operacji) z pełnej listy, którą `ssh -Q mac` pokazał jako teoretycznie dostępną.

TYLKO ODCZYT

## Pełna diagnostyka połączenia po stronie klienta: ssh -vvv

```
ssh -vvv nazwa_usera@NazwaSerwera
```

Trzy `v` to maksymalny poziom gadatliwości klienta — wypisujemy każdy etap negocjacji: którą wersję protokołu zaproponowano, które algorytmy kex/cipher/mac zostały uzgodnione, który klucz próbowano, dlaczego serwer go odrzucił. Pierwsze miejsce, gdzie szukać przyczyny, gdy logowanie kluczem „po prostu nie działa”, a komunikat końcowy klienta jest lakoniczny.

Przykład:

```
debug1: Offering public key: /home/adm_user/.ssh/id_ed25519 ED25519 SHA256:k3Jz...
debug1: Server accepts key: /home/adm_user/.ssh/id_ed25519 ED25519 SHA256:k3Jz...
debug1: Authentication succeeded (publickey).
```

**ssh** = Secure Shell: szyfrowane zdalne logowanie

TYLKO ODCZYT

## Test poprawności konfiguracji serwera: sshd -t

```
sudo sshd -t
```

Sprawdzamy składnię `/etc/ssh/sshd_config` (i plików dołączanych przez `Include`) bez uruchamiania czy restartowania usługi. Przy powodzeniu nie wypisuje nic — brak komunikatu jest tu dobrą wiadomością. Uruchomienie tego **zawsze** przed `systemctl restart sshd` jest jedynym rozsądnym nawykiem przy pracy na zdalnym serwerze.

> **Uważaj**
>
> - `ssh -Q` pyta klienta o jego możliwości; `sshd -T` pyta serwer o jego bieżącą, efektywną konfigurację — to nie jest ta sama odpowiedź i nie da się jednym zastąpić drugiego.
> - Restart `sshd` po błędnej zmianie konfiguracji może odciąć nas od serwera **bez żadnego ostrzeżenia w trakcie samego restartu** — usługa po prostu przestaje akceptować połączenia zgodnie z nową (błędną) polityką.
> - `ssh-agent` przechowuje odszyfrowane klucze tylko w pamięci bieżącej sesji — po zamknięciu terminala (bez `nohup`/menedżera sesji) agent i jego zawartość

znikają.

# Z życia wzięte

**Z ŻYCIA WZIĘTE**

O 9:30 Tomek pomaga nowemu koledze skonfigurować dostęp do docker01 kluczem SSH, ale kolega rozpakował klucz prywatny ze starego archiwum zip, które nie zachowało uprawnień. Logowanie odmawia użycia klucza z komunikatem o zbyt otwartych prawach, jeszcze zanim dojdzie do jakiejkolwiek próby połączenia z serwerem. Po `chmod 600` na pliku klucza logowanie przechodzi bez dalszych problemów.

# Po co to istnieje

**Pojęcie: para kluczy i kryptografia asymetryczna.** Kryptografia asymetryczna opiera się na dwóch matematycznie powiązanych liczbach (kluczach): tym, co zaszyfrowano lub podpisano jednym kluczem z pary, da się zweryfikować lub odszyfrować wyłącznie drugim — nigdy tym samym. Klucz publiczny jest bezpieczny do rozpowszechniania wszędzie, bo sam w sobie nie pozwala nikomu podszyć się pod właściciela; klucz prywatny musi pozostać tajny, bo to on jest dowodem tożsamości. W SSH ta para nie szyfruje treści logowania wprost — serwer wysyła losowe wyzwanie, a klient dowodzi posiadania klucza prywatnego, podpisując to wyzwanie w sposób, który da się zweryfikować kluczem publicznym zapisanym wcześniej w `authorized_keys`. Nigdzie po drodze klucz prywatny nie opuszcza Twojej maszyny.

**Pojęcie: odcisk klucza (fingerprint).** Klucz publiczny to zwykle kilkaset znaków tekstu — niewygodne do wzrokowego porównania. Odcisk to krótki, ustalonej długości skrót kryptograficzny (dziś domyślnie SHA-256) wyliczony z całego klucza, który zmienia się drastycznie przy najmniejszej zmianie oryginału. Służy do szybkiej, jednoznacznej weryfikacji „czy to na pewno ten sam klucz”, bez porównywania całej treści — dokładnie to widzisz przy pierwszym połączeniu z nowym hostem, kiedy klient SSH prosi o potwierdzenie odcisku klucza hosta.

Ten cały mechanizm istnieje, bo hasła mają fundamentalną słabość: da się je zgadnąć, złamać brute-force'em, wykraść przez phishing albo podejrzeć. Klucz prywatny o odpowiedniej długości jest praktycznie niemożliwy do odgadnięcia, a ponieważ nigdy nie jest przesyłany przez sieć (w

przeciwieństwie do hasła, nawet zaszyfrowanego kanałem TLS), nie da się go przechwycić po drodze — jedynym sposobem jego zdobycia jest dostęp do pliku, w którym leży na dysku. Stąd całe znaczenie uprawnień `600`/`700` opisanych niżej: to nie formalność, to cała bariera ochronna tego modelu.

> **Pojęcie: agent kluczy.** Klucz prywatny zabezpieczony hasłem (passphrase) wymaga jego podania przy każdym użyciu — niepraktyczne, jeśli łączysz się z dziesięcioma serwerami dziennie. Agent kluczy to osobny proces działający w tle Twojej sesji, który raz odszyfrowuje klucz do pamięci i od tej pory podpisuje w Twoim imieniu każde kolejne wyzwanie SSH, bez ponownego pytania o hasło. Klucz odszyfrowany trafia tylko do pamięci procesu agenta — nigdy nie jest zapisywany na dysku w formie odszyfrowanej.

> **Pojęcie: algorytm wymiany i MAC.** Każde połączenie SSH negocjuje trzy niezależne rodziny algorytmów: algorytm wymiany kluczy (kex) ustala wspólny tajny klucz sesji między klientem a serwerem bez przesyłania go wprost przez sieć; szyfr (cipher) tym kluczem szyfruje faktyczną treść przesyłanych danych; kod uwierzytelniania wiadomości (MAC) dolicza do każdego pakietu skrót gwarantujący, że po drodze nikt go nie zmienił, nawet jeśli sam atakujący nie zna klucza szyfrującego. Trzy osobne mechanizmy chronią przed trzema osobnymi zagrożeniami: podsłuchem klucza sesji, odczytaniem treści i po cichu zmodyfikowaniem danych w locie.

# Analogia

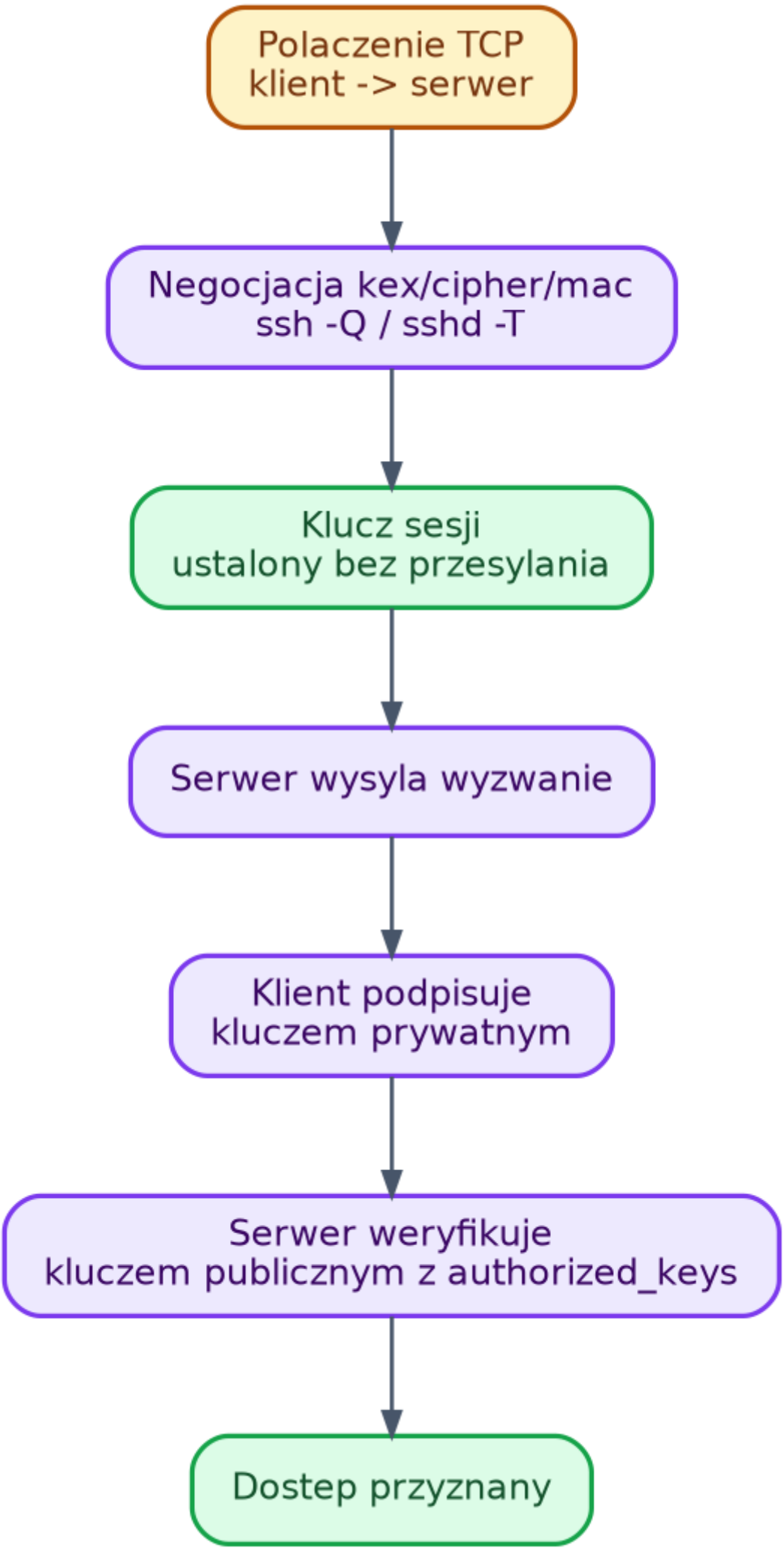
Wyobraź sobie **wzór podpisu złożony u notariusza**.

Zostawiasz w kancelarii swój wzór podpisu — coś, co każdy może obejrzeć, porównać, mieć u siebie w aktach. To jest Twój klucz publiczny. Nikt, kto ma dostęp do tego wzoru, nie jest w stanie na jego podstawie odtworzyć Twojej ręki na tyle dokładnie, żeby podrobić nowy, nigdy wcześniej niewidziany podpis pod nowym dokumentem — do tego potrzebna jest sama Twoja ręka, czyli klucz prywatny. Kiedy podpisujesz dokument, ktoś porównuje ten konkretny podpis ze wzorem w aktach i stwierdza zgodność, nie wiedząc nic więcej o mechanice Twojej ręki.

Zastrzeżenie, w którym analogia przestaje działać, jest ważne: podpis odręczny da się skopiować z dobrej fotokopii — matematyka klucza prywatnego temu zapobiega, dopóki plik z kluczem pozostaje wyłącznie w Twoich rękach. Ale odwróć to zastrzeżenie: jeśli ktoś zdobędzie nie fotokopię podpisu, tylko **samą Twoją rękę** — czyli plik klucza prywatnego — może odtąd składać podpisy nieodróżnialne od prawdziwych, w nieskończoność, bez żadnego śladu, że to nie Ty. Dlatego ochrona tego jednego pliku (uprawnienia, passphrase, ewentualnie sprzętowy nośnik) jest całym fundamentem zaufania, a nie formalnością do odhaczenia.

# Jak to działa

**Rozwiń: jak to działa pod spodem**


Polaczenie TCP
klient -> serwer
Negocjacja kex/cipher/mac
ssh -Q / sshd -T
Klucz sesji
ustalony bez przesylania
Serwer wysyla wyzwanie
Klient podpisuje
kluczem prywatnym
Serwer weryfikuje
kluczem publicznym z authorized_keys
Dostep przyznany

*Klient i serwer negocjują algorytmy i ustalają klucz sesji, potem serwer wysyła wyzwanie, klient podpisuje je kluczem prywatnym, a serwer weryfikuje podpis kluczem publicznym z authorized_keys (kliknięcie otwiera pełny rozmiar)*

Połączenie SSH przechodzi przez kilka etapów zanim dojdzie do sprawdzenia tożsamości użytkownika. Najpierw klient i serwer uzgadniają wersję protokołu i wspólną listę algorytmów — to właśnie ten etap opisują `ssh -Q kex/cipher/mac` (możliwości klienta) i `sshd -T` (efektywna, po uwzględnieniu `sshd_config`, konfiguracja serwera). Algorytm wymiany kluczy (np. `curve25519-sha256`) ustala tajny klucz sesji, którego nigdy nie trzeba przesyłać w postaci jawnej — obie strony wyliczają tę samą wartość z własnych, prywatnych fragmentów i publicznie wymienionych danych pośrednich. Dopiero na kanale zaszyfrowanym tym kluczem sesji odbywa się właściwe uwierzytelnienie użytkownika: serwer, mając Twój klucz publiczny w `authorized_keys`, wysyła losowe dane do podpisania, a Twój klient (albo `ssh-agent` w Twoim imieniu) podpisuje je kluczem prywatnym. Serwer weryfikuje podpis kluczem publicznym i wie, że rozmawia z posiadaczem właściwego klucza prywatnego — bez tego klucza nigdy nie widząc.

Dopiski `-etm@openssh.com` przy nazwach algorytmów MAC (encrypt-then-MAC) oznaczają nowszy, bezpieczniejszy porządek operacji: dane najpierw są szyfrowane, a dopiero zaszyfrowany wynik dostaje kod integralności. Starsze warianty bez tego dopisku liczyły MAC przed szyfrowaniem, co w pewnych konfiguracjach otwierało teoretyczne wektory ataku. Dlatego w wyniku `sshd -T` na zahartowanym serwerze produkcyjnym widać zwykle wyłącznie warianty `-etm`, mimo że `ssh -Q mac` pokazuje znacznie dłuższą, historyczną listę — administrator świadomie zawęził dozwolony zestaw w `sshd_config` przez dyrektywy `KexAlgorithms`, `Ciphers` i `MACs`.

Uprawnienia plików klucza są sprawdzane przez klienta SSH **przed** próbą użycia klucza, jeszcze zanim dojdzie do jakiejkolwiek negocjacji z serwerem — samo istnienie zbyt otwartego pliku wystarczy, żeby klient odmówił jego załadowania:

```
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
Permissions 0644 for '/home/user1/.ssh/id_ed25519' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
Load key "/home/user1/.ssh/id_ed25519": bad permissions
user1@srv02: Permission denied (publickey).
```

Po stronie serwera zasada jest analogiczna, tylko dla katalogu `.ssh` i pliku `authorized_keys` należącego do konta, na które ktoś próbuje się zalogować — `sshd`, zanim zaufa zapisanym tam kluczom publicznym, sprawdza, czy nikt poza właścicielem nie mógł ich podmienić. Jeśli katalog albo plik są zapisywalne przez grupę lub innych, serwer po cichu odrzuca uwierzytelnienie kluczem i przechodzi do kolejnej metody (zwykle hasła) — klient dostaje zwykłe `Permission denied (publickey)`, a prawdziwy powód widać dopiero w logu serwera:

```
Authentication refused: bad ownership or modes for directory /home/user1/.ssh
```

# Krok po kroku

### 1. Wygenerowanie klucza i sprawdzenie odcisku.

```
ssh-keygen -t ed25519 -f ~/.ssh/id_produkcja -C "adm_user@srv01"
ssh-keygen -lf ~/.ssh/id_produkcja.pub
```

```
256 SHA256:k3Jz9QpXv2mN8dL4wRt7YbHc1FgKs6TnWaEo0IqU5xA adm_user@srv01 (ED25519)
```

### 2. Instalacja klucza publicznego na serwerze.

```
ssh-copy-id -i ~/.ssh/id_produkcja.pub adm_user@srv01
```

### 3. Weryfikacja uprawnień po stronie serwera.

```
ssh adm_user@srv01 'ls -ld ~/.ssh ~/.ssh/authorized_keys'
```

```
drwx------ 2 adm_user adm_user 4096 wrz 17 10:02 /home/adm_user/.ssh
-rw------- 1 adm_user adm_user  412 wrz 17 10:02 /home/adm_user/.ssh/authorized_keys
```

`700` na katalogu, `600` na pliku — dokładnie to, co `ssh-copy-id` ustawił automatycznie.

### 4. Uruchomienie agenta i wygodne logowanie bez powtarzania passphrase.

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_produkcja
ssh-add -l
```

### 5. Skrót w `~/.ssh/config`, żeby nie pamiętać flag.

```
Host produkcja
    HostName srv01.ad.zielona.example
    User adm_user
    IdentityFile ~/.ssh/id_produkcja
```

```
ssh produkcja
```

**6. Diagnostyka, gdy logowanie kluczem nie działa.**

```
ssh -vvv produkcja 2>&1 | grep -A2 "Offering public key"
```

```
debug1: Offering public key: /home/adm_user/.ssh/id_produkcja ED25519 SHA256:k3Jz...
debug1: Authentications that can continue: publickey,password
debug1: No more authentication methods to try.
```

Ten konkretny fragment (klucz zaoferowany, ale serwer i tak przechodzi do listy „co dalej") to sygnał, że serwer **nie uznał** klucza za pasujący — czas sprawdzić `authorized_keys` i jego uprawnienia po stronie serwera, nie po stronie klienta.

**7. Hartowanie `sshd_config` — bez utraty dostępu.**

```
sudo sshd -t
```

Brak wyjścia oznacza poprawną składnię. Dopiero teraz, **z otwartą, działającą sesją SSH w tym samym oknie**, wprowadzasz zmiany:

```
PasswordAuthentication no
PermitRootLogin no
KexAlgorithms curve25519-sha256,curve25519-sha256@libssh.org
Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.com
MACs umac-128-etm@openssh.com,hmac-sha2-256-etm@openssh.com,hmac-sha2-512-etm@openssh.com
```

```
sudo sshd -t
sudo systemctl restart sshd
```

I dopiero po restarcie, **nie zamykając** obecnej sesji, otwierasz **nowe, osobne** połączenie w drugim oknie terminala, żeby potwierdzić, że nowa konfiguracja faktycznie pozwala się zalogować. Zamknięcie jedynej działającej sesji przed tym potwierdzeniem to najpewniejszy sposób na zablokowanie sobie dostępu do zdalnej maszyny.

# Częste błędy

**Zbyt otwarte uprawnienia klucza prywatnego.**

```
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
Permissions 0644 for '/home/user1/.ssh/id_ed25519' are too open.
Load key "/home/user1/.ssh/id_ed25519": bad permissions
```

```
user1@srv02: Permission denied (publickey).
```

Typowa przyczyna: klucz skopiowany przez `scp`/rozpakowany z archiwum, które nie zachowało oryginalnych uprawnień. Naprawa:

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/id_ed25519
```

**Zbyt otwarte uprawnienia po stronie serwera — klient nie widzi przyczyny.**

```
$ ssh user1@srv02
user1@srv02's password:
```

Klucz jest poprawny, uprawnienia po Twojej stronie w porządku, a serwer i tak prosi o hasło. Powodu nie zobaczysz w kliencie — trzeba zajrzeć do logu serwera:

```
Authentication refused: bad ownership or modes for directory /home/user1/.ssh
```

Naprawa (na serwerze, na koncie `user1`):

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
chown -R user1:user1 ~/.ssh
```

**Restart `sshd` po błędnej zmianie bez zapasowej sesji.**

Administrator zmienia `PermitRootLogin` albo `Port`, zapomina uruchomić `sshd -t`, i od razu robi `systemctl restart sshd` w tej samej, jedynej otwartej sesji. Jeśli konfiguracja ma błąd albo po prostu zamyka dotychczasową drogę dostępu, usługa albo w ogóle nie startuje, albo startuje z regułami, które odrzucają właśnie to połączenie — a stara sesja, w międzyczasie zerwana przez restart demona, już się nie odtworzy. Jedyne bezpieczne postępowanie: `sudo sshd -t` **zawsze** przed restartem, i restart wykonywany z zapasową, wciąż otwartą sesją, zamykaną dopiero po potwierdzeniu nowym, niezależnym połączeniem.

**Agent bez załadowanego klucza — powtarzające się pytanie o passphrase.**

```
Enter passphrase for key '/home/adm_user/.ssh/id_ed25519':
Enter passphrase for key '/home/adm_user/.ssh/id_ed25519':
```

`ssh-agent` uruchomiony, ale bez `ssh-add` jest po prostu pustym procesem — każde połączenie SSH i tak pyta o passphrase bezpośrednio, tak jakby agenta w ogóle nie było. Sprawdzenie `ssh-add -l` (`The agent has no identities.`) natychmiast pokazuje przyczynę; naprawa to `ssh-add ~/.ssh/id_ed25519`.

**Rozłączenie jump hosta traktowane jako błąd konfiguracji docelowego serwera.**

```
$ ssh -J bastion.ad.zielona.example user1@10.20.10.33
kex_exchange_identification: Connection closed by remote host
```

Ten komunikat pochodzi z etapu łączenia się z hostem pośredniczącym (`bastion.ad.zielona.example`), nie z docelowym serwerem wewnętrznym — administrator często zaczyna diagnozować niewłaściwą maszynę. Sprawdzenie: `ssh bastion.ad.zielona.example` osobno, bez `-J`, żeby ustalić, na którym z dwóch etapów połączenie faktycznie się rwie.

# □□ Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Klucze do samych ćwiczeń generujesz jako jednorazowe — nie używasz do nich prawdziwego klucza produkcyjnego.

## Ćwiczenie 1 — generowanie i odcisk

```
cd /tmp
ssh-keygen -t ed25519 -f /tmp/cwiczenie_klucz -N "" -C "cwiczenie"
ssh-keygen -lf /tmp/cwiczenie_klucz.pub
cat /tmp/cwiczenie_klucz.pub
```

Ile plików powstało i czym różni się zawartość jednego od drugiego pod względem tego, co wolno komukolwiek zobaczyć?

**Odpowiedź**

Powstają dwa pliki: `cwiczenie_klucz` (prywatny) i `cwiczenie_klucz.pub` (publiczny). Ten drugi jest bezpieczny do pokazania komukolwiek — to właśnie on trafia do `authorized_keys` na serwerach. Pierwszy nie powinien opuścić tej maszyny w żadnej formie; sama jego treść, gdyby wyciekła, pozwoliłaby podszyć się pod właściciela klucza bez żadnej możliwości wykrycia tego faktu przez ofiarę.

Sprzątanie:

```
rm /tmp/cwiczenie_klucz /tmp/cwiczenie_klucz.pub
```

# Ćwiczenie 2 — komunikat o złych uprawnieniach na własne oczy

```
cd /tmp
ssh-keygen -t ed25519 -f /tmp/zly_klucz -N "" -C "test-uprawnien"
chmod 644 /tmp/zly_klucz
ssh -i /tmp/zly_klucz -o IdentitiesOnly=yes localhost
```

(Jeśli lokalny `sshd` nie działa na Twojej maszynie, sam komunikat o złych uprawnieniach i tak pojawi się przed próbą połączenia sieciowego). Napraw i porównaj:

```
chmod 600 /tmp/zly_klucz
```

**Odpowiedź**

Przy `chmod 644` klient odmawia użycia klucza jeszcze przed jakąkolwiek próbą sieciową:

```
Permissions 0644 for '/tmp/zly_klucz' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
```

Po `chmod 600` ten sam plik jest akceptowany — klient przechodzi dalej do właściwej próby połączenia (która może się nie udać z innych powodów, np. braku działającego `sshd` na `localhost`, ale sam klucz nie jest już powodem odmowy).

Sprzątanie:

```
rm /tmp/zly_klucz /tmp/zly_klucz.pub
```

# Pytania kontrolne

**1. Dlaczego klucz publiczny można bezpiecznie wysłać e-mailem, a prywatnego nigdy?**

**Odpowiedź**

Kryptografia asymetryczna gwarantuje, że z klucza publicznego nie da się w rozsądnym czasie obliczeniowym odtworzyć klucza prywatnego. Klucz publiczny służy tylko do weryfikacji

podpisów złożonych kluczem prywatnym — jego ujawnienie nikomu nie pozwala podszyć się pod właściciela. Klucz prywatny to jedyny dowód tożsamości w tym systemie; jego wyciek pozwala trwale i niewykrywalnie podszywać się pod właściciela, dlatego musi zostać wyłącznie na maszynie, na której powstał (lub na zaufanym nośniku sprzętowym).

**2. Masz działającą sesję SSH na serwerze i chcesz zmienić `sshd_config`. Jaka jest jedna zasada, której złamanie grozi całkowitą utratą dostępu?**

**Odpowiedź**

Nigdy nie zamykaj jedynej otwartej sesji przed potwierdzeniem — nowym, niezależnym połączeniem — że zmieniona konfiguracja faktycznie pozwala się zalogować. `sshd -t` sprawdza tylko składnię, nie logikę biznesową zmiany (np. że właśnie wyłączyłeś jedyną metodę uwierzytelniania, z której korzystasz).

**3. `ssh -Q mac` pokazuje kilkanaście algorytmów, a `sudo sshd -T | grep '^macs'` na tym samym serwerze pokazuje tylko trzy. Co to oznacza?**

**Odpowiedź**

`ssh -Q mac` pokazuje pełne możliwości klienta OpenSSH — co teoretycznie potrafi zaproponować. `sshd -T` pokazuje efektywną, bieżącą konfigurację konkretnego serwera po uwzględnieniu `sshd_config`. Krótsza lista na serwerze oznacza, że administrator świadomie zawęził dozwolony zestaw algorytmów przez dyrektywę `MACs` — typowy element hartowania, zwykle ograniczający wybór do nowocześniejszych wariantów `-etm@openssh.com`.

**4. `ssh-add -l` zwraca `The agent has no identities.`, mimo że wcześniej uruchomiłeś `ssh-agent`. Co to oznacza i jaki jest następny krok?**

**Odpowiedź**

Agent działa, ale jest pusty — samo uruchomienie `ssh-agent` nie ładuje do niego żadnego klucza automatycznie. Następny krok to `ssh-add ~/.ssh/nazwa_klucza`, które odszyfrowuje wskazany klucz (pytając o passphrase, jeśli jest ustawiona) i przekazuje go działającemu agentowi.

**Powiązane:** sudo i /etc/sudoers: konfiguracja i audyt · Hasła i wygasanie kont: chage, passwd, faillock · Atrybuty rozszerzone: chattr, lsattr · Firewalld: strefy, usługi, rich rules

**Źródła:** `man 1 ssh` · `man 1 ssh-keygen` · `man 1 ssh-agent` · `man 1 ssh-copy-id` · `man 5 ssh_config` · `man 5 sshd_config` · `man 8 sshd` · OneNote „Ogolne" (SSH - klucz prywatny i publiczny per user;

Sprawdzenie SSH HMAC diagnostyka) · OneNote „Serwer SFTP" (uprawnienia katalogu `.ssh` i `authorized_keys`)

---

Revision #16
Created 2026-09-17 21:42:58 UTC by Tomasz Konieczny
Updated 2026-10-03 19:08:38 UTC by Tomasz Konieczny