Bezpieczeństwo systemów informatycznych

Zarządzanie kluczami

Marek Miśkiewicz

2025-11-19

UMCS Lublin

Zarządzanie kluczami

Źródła

NIST Special Publications (Rekomendacje Zarządzania Kluczami)

- Barker, E. (2020). “Recommendation for Key Management: Part 1 – General” (NIST Special Publication 800-57 Part 1 Revision 5, May 2020). Dostępne: https://doi.org/10.6028/NIST.SP.800-57pt1r5
- Barker, E., Barker, W. C. (2019). “Recommendation for Key Management: Part 2 – Best Practices for Key Management Organizations” (NIST Special Publication 800-57 Part 2 Revision 1, May 2019). Dostępne: https://doi.org/10.6028/NIST.SP.800-57pt2r1
- Barker, E., Dang, Q. H. (2015). “Recommendation for Key Management, Part 3: Application-Specific Key Management Guidance” (NIST Special Publication 800-57 Part 3, Rev. 1). Dostępne: https://doi.org/10.6028/NIST.SP.800-57pt3r1

Źródła

NIST Special Publications (Generowanie i Ustanawianie Kluczy)

- Barker, E., Roginsky, A. (2019). “Recommendation for Cryptographic Key Generation” (NIST Special Publication 800-133 Revision 1, July 2019). Dostępne: https://doi.org/10.6028/NIST.SP.800-133r1
- Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R. (2018). “Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography” (NIST Special Publication 800-56A, Rev. 3). Dostępne: https://doi.org/10.6028/NIST.SP.800-56Ar3
- Barker, E., Chen, L., Roginsky, A., Vassilev, A., Davis, R., Simon, S. (2019). “Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography” (NIST Special Publication 800-56B, Rev. 2). Dostępne: https://doi.org/10.6028/NIST.SP.800-56Br2
- Chen, L. (2009). “Recommendation for Key Derivation Using Pseudorandom Functions (Revised)” (NIST Special Publication 800-108). Dostępne: https://doi.org/10.6028/NIST.SP.800-108
- Barker, E., Kelsey, J. M. (2015). “Recommendation for Random Number Generation Using Deterministic Random Bit Generators” (NIST Special Publication 800-90A, Rev. 1). Dostępne: https://doi.org/10.6028/NIST.SP.800-90Ar1
- Barker, E., Smid, M. E., Branstad, D. K., Chokhani, S. (2013). “A Framework for Designing Cryptographic Key Management Systems” (NIST Special Publication 800-130). Dostępne: https://doi.org/10.6028/NIST.SP.800-130
- Barker, E. B., Branstad, D. K., Smid, M. E. (2015). “A Profile for U.S. Federal Cryptographic Key Management Systems (CKMS)” (NIST Special Publication 800-152). Dostępne: https://doi.org/10.6028/NIST.SP.800-152
- Barker, E. B. (2020). “Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms” (NIST Special Publication 800-175B, Rev. 1). Dostępne: https://doi.org/10.6028/NIST.SP.800-175Br1

Źródła

Federal Information Processing Standards (FIPS)

- National Institute of Standards and Technology. (2019). “Security Requirements for Cryptographic Modules” (Federal Information Processing Standards Publication 140-3). Dostępne: https://doi.org/10.6028/NIST.FIPS.140-3
- National Institute of Standards and Technology. (2015). “Secure Hash Standard (SHS)” (Federal Information Processing Standards Publication 180-4). Dostępne: https://doi.org/10.6028/NIST.FIPS.180-4
- National Institute of Standards and Technology. (2013). “Digital Signature Standard (DSS)” (Federal Information Processing Standards Publication 186-4). Dostępne: https://doi.org/10.6028/NIST.FIPS.186-4
- National Institute of Standards and Technology. (2001). “Advanced Encryption Standard (AES)” (Federal Information Processing Standards Publication 197). Dostępne: https://doi.org/10.6028/NIST.FIPS.197
- National Institute of Standards and Technology. (2008). “The Keyed-Hash Message Authentication Code (HMAC)” (Federal Information Processing Standards Publication 198-1). Dostępne: https://doi.org/10.6028/NIST.FIPS.198-1

Dlaczego zarządzanie kluczami jest kluczowe?

"Kryptografia bez odpowiedniego zarządzania kluczami jest jak pancerz ze stalowych drzwi, do których klucz leży pod wycieraczką"

Najczęstsze przyczyny kompromitacji systemów kryptograficznych:

- **Słabe generowanie kluczy** (przewidywalne źródła losowości)
- **Niezabezpieczone przechowywanie** (klucze w kodzie źródłowym, plaintext na dysku)
- **Brak rotacji kluczy** (używanie tego samego klucza przez lata)
- **Niekontrolowana dystrybucja** (przesyłanie przez niezabezpieczone kanały)
- **Brak procedur usuwania** (stare klucze pozostają w systemie)

Dodatkowe informacje

Cykl życia klucza kryptograficznego

7 faz zarządzania kluczem kryptograficznym

1. **Generowanie** - tworzenie klucza z bezpiecznego źródła entropii
2. **Dystrybucja** - bezpieczne przekazanie klucza uprawnionym stronom
3. **Przechowywanie** - ochrona klucza przed nieautoryzowanym dostępem
4. **Użycie** - wykorzystanie klucza zgodnie z przeznaczeniem
5. **Rotacja** - regularna wymiana kluczy
6. **Archiwizacja** - długoterminowe przechowywanie dla ciągłości biznesowej
7. **Niszczenie** - bezpieczne i nieodwracalne usunięcie klucza

Każda faza wymaga dedykowanych procedur bezpieczeństwa!

Cykl życia klucza kryptograficznego

1. Generowanie klucza

Źródła entropii:

- `/dev/urandom`, `/dev/random` (Linux/Unix)
- `CryptGenRandom` (Windows)
- Sprzętowe generatory liczb losowych (HRNG w procesorach Intel/AMD: RDRAND, RDSEED)

Wymagania:

- Kryptograficznie bezpieczny generator pseudolosowy (CSPRNG)
- Wystarczająca entropia (co najmniej tyle bitów, ile długość klucza – np. 128 bitów dla AES-128, 256 bitów dla AES-256)
- Niemożliwość przewidzenia kolejnych wartości na podstawie poprzednich

Generacja klucza - przykłady

Złe praktyki:

```
# NIGDY TAK NIE RÓB!
import random
key = random.randint(0, 2**256)  # NIE jest kryptograficznie bezpieczne
```

Dobre praktyki:

```
# Python - prawidłowe generowanie klucza
import secrets
key = secrets.token_bytes(32)  # 256 bitów

# Lub z wykorzystaniem biblioteki cryptography
from cryptography.hazmat.primitives.ciphers import algorithms
key = algorithms.AES.generate_key(256)
```

2. Dystrybucja kluczy

Problem wymiany kluczy - jak bezpiecznie przekazać klucz stronie, z którą chcemy się komunikować?

Jest to jeden z fundamentalnych problemów w kryptografii symetrycznej. Jeśli Alice i Bob chcą komunikować się bezpiecznie, jak mogą wymienić klucz przez niezabezpieczony kanał?

Dystrybucja kluczy - rozwiązania

1. Pre-shared keys (PSK):

- Klucz ustalony wcześniej przez bezpieczny kanał
- Osobiste spotkanie, kurier, telefon
- Stosowane w: VPN, WiFi (WPA2/WPA3-PSK)

Zalety:

- Prostota
- Wysoka wydajność (tylko kryptografia symetryczna)

Wady:

- Wymaga bezpiecznego kanału do początkowej wymiany
- Problemy ze skalowalnością (n użytkowników → $\frac{n(n-1)}{2}$ kluczy)
- Trudna rotacja kluczy

Dystrybucja kluczy - rozwiązania

2. Kryptografia asymetryczna:

Wykorzystanie algorytmów z kluczem publicznym do bezpiecznej wymiany klucza symetrycznego.

Schemat Diffie-Hellman (DH, ECDH):

- Dwie strony ustalają wspólny sekret przez publiczny kanał
- Sekret nie jest nigdy przesyłany
- Matematyka: problem logarytmu dyskretnego

Szyfrowanie klucza asymetrycznego:

- Alice szyfruje klucz symetryczny kluczem publicznym Boba (RSA-OAEP)
- Tylko Bob może go odszyfrować swoim kluczem prywatnym

Dystrybucja kluczy - rozwiązania

3. Key Derivation Functions (KDF):

Wyprowadzanie wielu kluczy z jednego klucza głównego lub **materiału klucza**.

Materiał klucza (Key Material): surowe dane o wysokiej entropii, które mogą zostać przekształcone w klucze kryptograficzne. Może to być:

- Wspólny sekret z wymiany Diffie-Hellman
- Hasło użytkownika
- Istniejący klucz kryptograficzny (master key)
- Dane losowe z CSPRNG

Zastosowania:

- Wyprowadzanie kluczy sesji z master key
- Generowanie różnych kluczy do różnych celów z jednego sekretu
- Protokoły: TLS 1.3 (z DH exchange → klucze sesji), Signal Protocol, IPsec

HMAC i bcrypt

HMAC - Hash-based Message Authentication Code

Definicja: HMAC to kryptograficzna funkcja używana do weryfikacji integralności i autentyczności danych.

Mechanizm: $\text{HMAC}(K, m) = H((K \oplus \text{opad}) \parallel H((K \oplus \text{ipad}) \parallel m))$

gdzie:

- K - klucz tajny
- m - wiadomość
- H - funkcja hash (np. SHA-256)
- opad, ipad - stałe padding values (outer, inner)

Właściwości:

- Wymaga klucza tajnego (w przeciwieństwie do zwykłego hash)
- Deterministyczny: ten sam klucz + wiadomość → ten sam HMAC
- Jednokierunkowy: nie można odtworzyć klucza ani wiadomości z HMAC
- Odporny na kolizje (dzięki użyciu funkcji hash)

HMAC - zastosowania

Gdzie używany?

1. **Weryfikacja integralności**:
   - Upewnienie, że wiadomość nie została zmodyfikowana
   - Nadawca: oblicza `HMAC(key, message)` i wysyła razem z wiadomością
   - Odbiorca: ponownie oblicza HMAC i porównuje
2. **Uwierzytelnianie API** (np. AWS Signature):

```
signature = HMAC-SHA256(secret_key, request_data)
# Wysyłane w nagłówku: Authorization: AWS4-HMAC-SHA256 ...
```

HMAC - zastosowania

Gdzie używany?

3. **Protokoły kryptograficzne**:
   - TLS/SSL (weryfikacja handshake)
   - IPsec (autentykacja pakietów)
   - JWT (JSON Web Tokens) - podpis tokenu
4. **Key Derivation Functions** (KDF):
   - PBKDF2 używa HMAC jako pseudo-random function (PRF)
   - HKDF bazuje na HMAC do wyprowadzania kluczy

Password Hashing vs Key Derivation - różnice

Często mylone koncepcje!

	Password Hashing Function	Key Derivation Function (KDF)
Cel:	Weryfikacja tożsamości	Generowanie kluczy kryptograficznych
Input:	Hasło użytkownika	Hasło lub materiał klucza
Output:	Hash (nie-deterministyczny z embedded salt)	Deterministyczny klucz o określonej długości
Użycie:	`hash(password) == stored_hash`	Klucz do AES, HMAC, itp.
Przykłady:	bcrypt, scrypt (dla haseł)	PBKDF2, scrypt (dla kluczy), Argon2, HKDF

Password Hashing vs Key Derivation - różnice

Kiedy używać czego?

Password Hashing (bcrypt, scrypt dla haseł):

- Przechowywanie haseł w bazie danych
- Uwierzytelnianie użytkowników (login)
- Nie-deterministyczny output (różne hashe dla tego samego hasła)
- Wbudowana sól w wyniku

Key Derivation (PBKDF2, Argon2, HKDF):

- Generowanie klucza AES z hasła użytkownika
- Wyprowadzanie kluczy sesji z master key
- Deterministyczny output (to samo hasło + sól → ten sam klucz)
- Kontrola nad długością klucza

UWAGA: scrypt i Argon2 to funkcje konfigurowalne – w zależności od parametrów mogą służyć zarówno jako funkcje hashujące hasła, jak i KDF.

bcrypt - Password Hashing Function

Specyfika: bcrypt to **password hashing function** służąca do **weryfikacji haseł**, nie do generowania kluczy kryptograficznych.

Mechanizm:

- Oparty na szyfrze Blowfish (adaptacyjna funkcja kryptograficzna)
- Iteracyjne hashowanie z użyciem algorytmu Eksblowfish
- Wbudowana sól (salt) w wyniku

Główna cecha: **Cost factor** (work factor)

- Parametr określający liczbę iteracji: 2^{cost}
- Domyślnie: cost = 10 → 2^{10} = 1024 iteracji
- Zwiększenie cost o 1 → podwojenie czasu obliczenia
- Adaptacyjność: można zwiększać cost wraz z rozwojem sprzętu

Password Hashing - bcrypt

Format wyniku:

```
$2b$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
|  |  |                      |
|  |  |                      └─ Hash (31 znaków)
|  |  └─ Sól (22 znaki, Base64)
|  └─ Cost factor
└─ Wersja bcrypt (2a, 2b, 2y)
```

Password Hashing - bcrypt

Przykład użycia bcrypt:

```
import bcrypt

# Hashowanie hasła
password = b"super_secret_password"
salt = bcrypt.gensalt(rounds=12)  # cost = 12 → 2^12 = 4096 iteracji
hashed = bcrypt.hashpw(password, salt)

# Weryfikacja hasła
if bcrypt.checkpw(password, hashed):
    print("Hasło poprawne")
```

Password Hashing - bcrypt

Zalecane parametry (OWASP 2024):

- Cost factor: minimum 10 (zalecane 12-14)
- Im wyższy cost, tym dłuższy czas (testuj na docelowym sprzęcie)
- Cel: 100-500ms na weryfikację hasła

Zalety bcrypt:

- Prosta w użyciu (wbudowana sól)
- Sprawdzona od 1999 roku
- Szeroko wspierana w językach programowania
- Odporność na ataki GPU (w pewnym stopniu)

Wady bcrypt:

- Ograniczenie długości hasła: max 72 bajty (!)
  - Dłuższe hasła są obcinane
  - Workaround: hashuj hasło (SHA-256) przed bcrypt
- Mniej odporna na GPU niż scrypt/Argon2
- Brak parametru pamięci (tylko CPU cost)

Uwaga: bcrypt można technicznie użyć do KDF, ale nie jest to jego główne przeznaczenie!

Key Derivation Functions (KDF)

Wprowadzenie do KDF

KDF dzielimy na dwie główne kategorie:

1. Password-based KDF:
- Przekształcają słabe hasło → silny klucz kryptograficzny
- Wymagają **slow hash** (ochrona przed brute-force)
- Przykłady: PBKDF2, scrypt, Argon2
- Zastosowanie: szyfrowanie dysków, plików, generowanie kluczy z hasła

2. Key-based KDF:
- Przekształcają materiał o wysokiej entropii → uporządkowane klucze
- Mogą być szybkie (materiał już ma wysoką entropię)
- Przykłady: HKDF, ANSI X9.63 KDF
- Zastosowanie: wyprowadzanie kluczy sesji w TLS, Signal

PBKDF2 (Password-Based Key Derivation Function 2)

Mechanizm:

- Iteracyjne hashowanie hasła z solą

$$DK = PBKDF2(PRF, Password, Salt, c, dkLen)$$

Parametry:

- `PRF` (Pseudo-Random Function): funkcja kryptograficzna, zazwyczaj HMAC-SHA256
- `Password`: hasło użytkownika (surowy materiał wejściowy)
- `Salt`: losowa wartość (min. 128 bitów) - zapobiega rainbow table attacks
- `c` (count): liczba iteracji - zwiększa czas obliczenia
  - Zalecane min. 600,000 dla SHA-256 (OWASP 2023)
  - Dostosować do mocy obliczeniowej: 100ms czasu na urządzeniu docelowym
- `dkLen`: długość wynikowego klucza (np. 256 bitów dla AES-256)

Key Derivation Functions (KDF)

Przykład:

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes

kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=600000,
)
key = kdf.derive(password)
```

Password-based KDF - scrypt

Mechanizm:

- Memory-hard function: algorytm wymaga dużej ilości pamięci RAM do wykonania
- Przewaga: pamięć RAM jest droga i trudna do zrównoleglenia
- Utrudnia ataki sprzętowe (GPU, ASIC, FPGA) - te urządzenia mają ograniczoną pamięć

Parametry:

- `N`: CPU/memory cost factor (koszt obliczeniowy, potęga 2)
  - Im wyższe N, tym więcej pamięci i czasu potrzeba
- `r`: block size (rozmiar bloku, wpływa na użycie pamięci)
- `p`: parallelization factor (stopień równoległości)

Key Derivation Functions (KDF)

Zalecane parametry (OWASP 2024):

- **Interaktywne** (logowanie użytkownika): $N=2^{16}$ (65536), r=8, p=1
  - Wymaga 64 MB RAM, 100ms czasu
- **Nieinteraktywne** (szyfrowanie plików): $N=2^{20}$ (1048576), r=8, p=1
  - Wymaga 1 GB RAM, może trwać kilka sekund

Zastosowania:

- Litecoin (proof-of-work w blockchain)
- Tarsnap (backup z szyfrowaniem)

Password-based KDF - Argon2

Zwycięzca Password Hashing Competition (2015)

Trzy warianty:

- **Argon2d**: odporny na time-memory trade-off, podatny na timing attacks
  - Nie można "oszukać" i użyć mniej pamięci kosztem dodatkowych obliczeń
  - Atakujący MUSI użyć pełnej ilości pamięci przy każdym sprawdzeniu hasła
- **Argon2i**: odporny na timing attacks, mniej odporny na trade-off
- **Argon2id**: hybrydowy, **zalecany** (najlepsze połączenie obu technik)

Parametry:

- Pamięć (m): min. 19 MB dla ochrony logowania
- Iteracje (t): min. 2
- Równoległość (p): dostosowane do liczby rdzeni

Zalety:

- Najnowocześniejszy standard (OWASP zaleca dla nowych implementacji)
- Odporny na GPU/ASIC attacks
- Dobra odporność na side-channel attacks

Key Derivation Functions (KDF)

Pamięć jest dużo droższa niż obliczenia dla atakującego:

- GPU/ASIC mają tysiące rdzeni obliczeniowych (tanie)
- Ale mają ograniczoną pamięć (droga)

Przykład:

- GPU może liczyć 1000 hashy PBKDF2 równolegle (każdy w osobnym rdzeniu)
- Ale jeśli każdy hash Argon2 wymaga 512 MB RAM → GPU z 16 GB może liczyć tylko 32 hashe jednocześnie

Password-based KDF - porównanie

Kiedy używać którego algorytmu?

	PBKDF2	scrypt	Argon2id
Rok	2000 (RFC 2898)	2009	2015
Ochrona	CPU cost	CPU + memory cost	CPU + memory cost + timing
Odporność GPU	Niska	Średnia	Wysoka
Zalecenia OWASP	600k iteracji SHA-256	$N=2^{16}$, r=8, p=1	m=19MB, t=2, p=1
Kiedy używać	Legacy systems, szeroka kompatybilność	Jeśli Argon2 niedostępny	Nowe implementacje (preferowany)
Wsparcie	Wszędzie	Dobre	Rosnące
Prędkość	Najszybszy	Wolniejszy	Konfigurowalny

Rekomendacja: Argon2id dla nowych systemów, PBKDF2 jako fallback dla kompatybilności.

Key-based KDF - HKDF (HMAC-based KDF)

Przeznaczenie: wyprowadzanie kluczy z materiału klucza o wysokiej entropii (NIE z hasła!)

Dwa etapy:

1. **Extract**: koncentracja entropii
   - PRK = HMAC-Hash(salt, IKM)
   - IKM (Input Keying Material) - materiał źródłowy o wysokiej entropii:
     - Wspólny sekret z wymiany Diffie-Hellman/ECDH
     - Istniejący klucz kryptograficzny (master key)
     - Surowe dane losowe z CSPRNG
     - Dane z hardware RNG
   - PRK (Pseudo-Random Key) - pośredni klucz o równomiernej entropii

Key Derivation Functions (KDF)

2. **Expand**: generacja wielu kluczy
   - OKM = HKDF-Expand(PRK, info, L)
   - PRK (Pseudo-Random Key) - klucz z etapu Extract
   - info - kontekst aplikacji (np. “AES-encryption-key”, “HMAC-key”)
   - L - wymagana długość klucza wyjściowego
   - OKM (Output Keying Material) - finalny klucz lub klucze

Zastosowania:

- TLS 1.3 (wyprowadzanie kluczy sesji)
- Signal Protocol, Noise Protocol Framework

3. Przechowywanie kluczy

Najbardziej krytyczny element zarządzania kluczami!

Klucz musi być chroniony przed:

- Nieautoryzowanym dostępem
- Przypadkowym ujawnieniem (logi, core dumps, swap)
- Kradzieżą (malware, insider threat)
- Utratą (awaria sprzętu, przypadkowe usunięcie)

Przechowywanie kluczy - poziomy bezpieczeństwa

Poziom 0 - NIGDY!

```
# Hardcoding w kodzie źródłowym
SECRET_KEY = "my_super_secret_key_12345"
API_KEY = "sk-1234567890abcdef"
DB_PASSWORD = "admin123"
```

Problemy:

- Klucz widoczny w repozytorium (Git, SVN)
- Dostępny dla każdego z dostępem do kodu
- Historia commitów zachowuje stare klucze
- Trudna rotacja (wymaga zmiany kodu)

Przykład z życia: Uber 2016 - klucze AWS w GitHub → $148 mln kary

Przechowywanie kluczy - poziomy bezpieczeństwa

Poziom 1 - Zmienne środowiskowe (niezalecane)

```
export DATABASE_KEY="..."
export API_SECRET="..."
```

```
import os
secret_key = os.environ.get('SECRET_KEY')
```

Zalety:

- Klucz poza kodem źródłowym
- Łatwa zmiana bez modyfikacji kodu

Wady:

- Widoczne w `ps aux`, `/proc/PID/environ`
- Mogą trafić do logów
- Dziedziczone przez procesy potomne
- Przypadkowo eksportowane w core dumps

Przechowywanie kluczy - poziomy bezpieczeństwa

Poziom 2 - Pliki konfiguracyjne z odpowiednimi uprawnieniami

```
# Właściwe uprawnienia
chmod 600 /etc/app/secrets.conf
chown app:app /etc/app/secrets.conf
# ls -la
-rw------- 1 app app 256 secrets.conf  # DOBRZE - tylko właściciel
```

```
# Odczyt z chronionego pliku
with open('/etc/app/secrets.conf', 'r') as f:
    secret = f.read().strip()
```

Zalety:

- Kontrola dostępu na poziomie systemu operacyjnego
- Audyt dostępu (auditd, SELinux)

Wady:

- Nadal plaintext na dysku
- Podatne na rootkity i eskalację uprawnień

Przechowywanie kluczy - poziomy bezpieczeństwa

Poziom 3 - Systemy zarządzania sekretami

HashiCorp Vault:

- Centralne zarządzanie sekretami
- Szyfrowanie at-rest i in-transit
- Dynamiczne sekrety (generowane na żądanie, krótki TTL)
- Audit logging
- Sealing/unsealing (klucz do odszyfrowania Vault)

Cloud KMS:

- **AWS Secrets Manager / KMS**: integracja z IAM, automatyczna rotacja
- **Azure Key Vault**: integracja z Azure AD, HSM-backed
- **Google Cloud KMS**: integracja z Cloud IAM, multi-region

Kubernetes Secrets:

- Przechowywanie w etcd (szyfrowane at-rest)
- Montowanie jako woluminy lub zmienne środowiskowe
- RBAC dla kontroli dostępu

Przechowywanie kluczy - poziomy bezpieczeństwa

Przykład z HashiCorp Vault:

Vault to system do bezpiecznego przechowywania i zarządzania sekretami (klucze API, hasła do baz danych, certyfikaty).

```
# 1. Administrator zapisuje sekret w Vault
vault kv put secret/myapp/config \
    api_key="sk-1234567890abcdef" \
    db_password="super_secure_password_xyz"

# 2. Aplikacja odczytuje sekret (wymaga autoryzacji)
vault kv get -field=api_key secret/myapp/config
# Output: sk-1234567890abcdef

# 3. W aplikacji Python
import hvac
client = hvac.Client(url='https://vault.example.com', token=os.environ['VAULT_TOKEN'])
secret = client.secrets.kv.v2.read_secret_version(path='myapp/config')
api_key = secret['data']['data']['api_key']
```

Przechowywanie kluczy - poziomy bezpieczeństwa

Zalety Vault:

- Centralizacja (jeden punkt zarządzania dla całej organizacji)
- Rotacja automatyczna (klucze mogą być automatycznie zmieniane co X dni)
- Wersjonowanie sekretów (historia zmian, możliwość rollback)
- Audit trail (kto, kiedy, jaki sekret odczytał - compliance)
- Dynamic secrets (Vault generuje tymczasowe hasło do bazy na 24h, potem automatycznie kasuje)

Przechowywanie kluczy - poziomy bezpieczeństwa

Poziom 4 - Hardware Security Modules (HSM)

Najwyższy poziom ochrony kluczy kryptograficznych

Cechy HSM:

- Dedykowane urządzenia sprzętowe
- Klucz NIGDY nie opuszcza HSM
- Wszystkie operacje kryptograficzne wewnątrz modułu
- Certyfikowane (FIPS 140-2/3 Level 3 lub 4)
- Ochrona fizyczna:
  - Tamper-evident (wykrywa próby ingerencji)
  - Tamper-resistant (opiera się próbom ataku)
  - Tamper-responsive (automatycznie kasuje klucze przy ataku)

Hardware Security Modules (HSM)

Typy HSM

1. Sieciowe HSM:

- Przykłady: Thales Luna, nCipher nShield, Utimaco SecurityServer
- Współdzielone przez wiele aplikacji
- Wysokowydajne (tysiące operacji/sekundę)
- Redundancja i high availability
- Koszt: $20,000 - $100,000+

2. HSM w chmurze:

- AWS CloudHSM, Azure Dedicated HSM, Google Cloud HSM
- Dedykowany HSM w infrastrukturze dostawcy
- Pełna kontrola klienta nad kluczami
- Single-tenant (jeden klient = jeden HSM)
- Koszt: $1,000 - $2,000/miesiąc

Hardware Security Modules (HSM)

3. USB/PCIe HSM:

- Przykłady: YubiHSM 2, Nitrokey HSM
- Dla pojedynczych stacji roboczych/serwerów
- Niższy koszt ($50 - $500)
- Ograniczona wydajność

4. Trusted Platform Module (TPM):

- Chip wbudowany w płytę główną
- TPM 2.0 - aktualny standard
- Przechowywanie kluczy szyfrujących dysk (BitLocker, LUKS)
- Poświadczanie integralności systemu (Secure Boot, measured boot)
- Attestation (dowód, że system nie został zmodyfikowany)

Zastosowania HSM

1. Infrastruktura PKI (Public Key Infrastructure)

Certificate Authority (CA) wystawia certyfikaty dla całego internetu (np. Let's Encrypt, DigiCert). Klucz prywatny Root CA jest najbardziej krytyczny - jeśli zostanie skompromitowany, atakujący może podszywać się pod dowolną stronę WWW. HSM zapewnia, że ten klucz nigdy nie opuści bezpiecznego modułu. Wszystkie operacje podpisywania certyfikatów odbywają się wewnątrz HSM.

2. Transakcje finansowe

Banki i systemy płatnicze przetwarzają miliony transakcji dziennie. PCI DSS (Payment Card Industry Data Security Standard) wymaga użycia HSM do ochrony PIN-ów kart płatniczych. HSM szyfruje PIN przy wypłacie z bankomatu i weryfikuje go podczas płatności kartą. EMV (chip cards) używa HSM do generowania kluczy dla chipów w kartach.

3. Blockchain i kryptowaluty

Instytucje finansowe (fundusze hedgingowe, giełdy kryptowalut) zarządzają portfelami wartymi miliardy dolarów. Klucz prywatny Bitcoin/Ethereum to jedyny sposób dostępu do środków - jego utrata = trwała utrata pieniędzy. Custody solutions (Coinbase Custody, Fidelity Digital Assets) używają HSM do przechowywania kluczy prywatnych. Transakcja jest podpisywana wewnątrz HSM, klucz nigdy nie jest eksportowany.

4. Code signing (podpisywanie kodu)

Producenci oprogramowania podpisują swoje programy, aby użytkownicy mogli zweryfikować autentyczność (np. Windows sprawdza podpis przed instalacją). Jeśli klucz code-signing zostanie skradziony, atakujący może podpisać malware jako zaufane oprogramowanie (przykład: atak SolarWinds 2020). HSM chroni klucze code-signing przed kradzieżą.

5. Telekomunikacja

Operatorzy sieci 5G muszą szyfrować komunikację dla milionów abonentów. Każdy telefon ma unikalny klucz, a operacje kryptograficzne (autentykacja, szyfrowanie połączeń) odbywają się w HSM w centrali operatora. HSM musi być wysokowydajny - tysiące operacji na sekundę dla milionów użytkowników.

Certyfikacja FIPS 140-2/3

FIPS = Federal Information Processing Standards (amerykański standard rządowy NIST)

Standard definiuje cztery poziomy bezpieczeństwa dla modułów kryptograficznych:

Level 1: Podstawowa ochrona kryptograficzna
- Może być implementacja software'owa (np. biblioteka OpenSSL)
- Brak wymagań fizycznych
- Przykład: cryptography w Pythonie, OpenSSL

Level 2: Ochrona fizyczna + kontrola dostępu
- Tamper-evident: wykrywa próby ingerencji (plomby, naklejki)
- Role-based authentication: różne role (admin, operator, audytor)
- Przykład: YubiKey FIPS, podstawowe USB HSM

Hardware Security Modules (HSM)

Level 3: Aktywna ochrona przed atakami

- **Tamper-resistant**: opiera się atakom (wzmocniona obudowa, mesh detector)
- **Identity-based authentication**: uwierzytelnianie konkretnych osób (nie tylko ról)
- Większość komercyjnych HSM (Thales Luna, AWS CloudHSM)
- Wymagane przez PCI DSS Level 3+ dla przetwarzania płatności

Level 4: Najwyższy poziom (wojsko, rząd)

- **Tamper-responsive**: automatycznie kasuje klucze przy próbie ataku
- **Environmental protection**: ochrona przed napięciem, temperaturą, promieniowaniem
- Przykład: HSM w łączności rządowej, obiekty wojskowe

Dlaczego certyfikacja jest ważna?

- Wiele regulacji wymaga FIPS (PCI DSS, HIPAA, FedRAMP)
- Certyfikacja kosztuje $100k$ –500k i trwa 12-18 miesięcy
- Gwarancja niezależnego audytu bezpieczeństwa

4. Użycie klucza

Zasady bezpiecznego użycia kluczy

1. Separation of duties (rozdzielenie obowiązków):

- Różne klucze do różnych celów
- Klucz do szyfrowania ≠ klucz do podpisywania
- Klucz produkcyjny ≠ klucz testowy
- Klucz dla aplikacji A ≠ klucz dla aplikacji B

2. Principle of least privilege:

- Dostęp tylko dla uprawnionych procesów/użytkowników
- Minimalizacja czasu, przez który klucz jest w pamięci
- Kasowanie klucza z pamięci po użyciu (zerowanie)

Użycie klucza - Key Wrapping

Key wrapping - szyfrowanie klucza innym kluczem

Terminologia:

- **KEK** (Key Encryption Key): klucz służący do szyfrowania innych kluczy
- **DEK** (Data Encryption Key): klucz używany do szyfrowania danych

Hierarchia kluczy:

```
Master Key (w HSM)
    ↓ (szyfruje)
Key Encryption Keys (KEK)
    ↓ (szyfruje)
Data Encryption Keys (DEK)
    ↓ (szyfruje)
Zaszyfrowane dane
```

Użycie klucza - Key Wrapping

Korzyści hierarchii:

- Master key może być przechowywany w HSM (nigdy nie opuszcza)
- KEK mogą być dystrybuowane (zaszyfrowane przez master key)
- DEK mogą być przechowywane w bazie danych (zaszyfrowane przez KEK)
- Rotacja DEK nie wymaga odszyfrowania wszystkich danych

Co się dzieje gdy wycieknie zaszyfrowany DEK?

Załóżmy scenariusz: atakujący uzyskuje dostęp do bazy danych i kradnie zaszyfrowany DEK.

Hierarchia ochrony:

```
Master Key (HSM - fizycznie chroniony)
    ↓ szyfruje
KEK (w pamięci aplikacji, zaszyfrowany przez Master Key)
    ↓ szyfruje
DEK (w bazie danych, zaszyfrowany przez KEK) ← WYCIEKŁ!
    ↓ szyfruje
Dane (w bazie danych, zaszyfrowane przez DEK)
```

Użycie klucza - Key Wrapping

Dlaczego zaszyfrowany DEK jest bezużyteczny?

1. **Atakujący ma**: zaszyfrowany DEK, zaszyfrowane dane
2. **Atakujący NIE MA**: KEK (jest w pamięci aplikacji) ani Master Key (jest w HSM)
3. **Rezultat**: nie może odszyfrować DEK, więc nie może odszyfrować danych

Porównanie z pojedynczym kluczem:

Gdyby dane były szyfrowane bezpośrednio jednym kluczem przechowywanym w bazie:

- Wyciek bazy = wyciek klucza = natychmiastowa kompromitacja danych
- Envelope encryption wymaga kompromitacji wielu warstw jednocześnie

Dlaczego to działa?

- Każda warstwa ma inny wektor ataku (baza danych ≠ pamięć aplikacji ≠ HSM)
- Kompromitacja jednej warstwy nie daje dostępu do innych
- Defense in depth: atakujący musi sforsować wszystkie warstwy

Standard AES Key Wrap (RFC 3394):

- Specjalny tryb AES do bezpiecznego opakowywania kluczy
- Zapewnia integralność i autentyczność oprócz poufności
- Używany w: TLS, IPsec, Kerberos

Użycie klucza - Audit logging

Rejestrowanie wszystkich operacji na kluczach

Co logować:

- Kto uzyskał dostęp? (user, process, IP address)
- Kiedy? (timestamp z dokładnością do milisekund)
- Do jakiego klucza? (key ID, nie sam klucz!)
- Jaka operacja? (generate, encrypt, decrypt, sign, verify, export)
- Rezultat? (success, failure, error code)

NIGDY nie loguj:

- Samego klucza
- Danych przed/po szyfrowaniu
- Hasła użytkownika

Przechowywanie logów:

- Niezmienialne logi (append-only, WORM storage)
- Ochrona przed modyfikacją (podpis cyfrowy, blockchain)
- Długoterminowa retencja (zgodnie z polityką/regulacjami)

5. Rotacja kluczy

Dlaczego rotacja jest ważna?

- **Ograniczenie szkód**: kompromitacja klucza wpływa tylko na dane z okresu jego użycia
- **Zmniejszenie kryptoanalizy**: mniej danych zaszyfrowanych jednym kluczem
- **Compliance**: zgodność z politykami bezpieczeństwa i regulacjami
- **Zmniejszenie ryzyka**: im dłużej klucz jest używany, tym większe prawdopodobieństwo kompromitacji

Cryptographic lifetime vs **Operational lifetime**:

- Cryptographic: ile danych można bezpiecznie zaszyfrować
- Operational: jak długo klucz powinien być używany w praktyce

Rotacja kluczy - częstotliwość

Zalecenia NIST (SP 800-57)

Klucze symetryczne (np. AES):

- Co 1-2 lata lub
- Po zaszyfrowaniu $2^{\frac{n}{2}}$ bloków (birthday bound)
- AES-128: 2^{64} bloków = 2^{68} bajtów ≈ 256 exabajtów (czyli 262 000 petabajtów)
- W praktyce: rotacja znacznie wcześniej (roczna/półroczna)

Klucze asymetryczne:

- Klucz prywatny CA: 5-10 lat (Root CA), 2-5 lat (Intermediate CA)
- Certyfikaty SSL/TLS: 1-2 lata (obecnie często 90 dni - Let's Encrypt, ZeroSSL)
- Klucze SSH: rocznie lub po zdarzeniu bezpieczeństwa
- Klucze code signing: 1-3 lata

Rotacja kluczy - strategie

1. Blue-Green rotation:

- Przygotuj nowe środowisko (green) z nowym kluczem
- Przełącz całą nową komunikację/operacje na środowisko green (cut-over)
- Opcjonalnie pozostaw środowisko blue tylko do odczytu lub rollbacku przez krótki czas
- Po okresie przejściowym usuń lub zdezaktywuj stary klucz (blue)

Zalety:

- Szybkie i jednoznaczne przełączenie na nowy klucz
- Czytelna możliwość rollbacku (powrót do blue, jeśli coś pójdzie źle)
- Brak długotrwałej koegzystencji wielu kluczy dla tych samych danych

Wady:

- Wymaga utrzymywania dwóch środowisk jednocześnie (blue i green)
- Trudniejsze w systemach rozproszonych i przy bardzo dużych wolumenach danych

Rotacja kluczy - strategie

2. Lazy rotation / Opportunistic re-encryption:

- Nowe dane szyfrowane nowym kluczem
- Stare dane pozostają zaszyfrowane starym kluczem
- Przy odczycie: odszyfrowanie starym, ponowne szyfrowanie nowym

Zalety:

- Natychmiastowa rotacja dla nowych danych
- Nie wymaga przestoju
- Stopniowa migracja w tle

Wady:

- Stare dane mogą pozostać ze starym kluczem długo
- Dane read-only nigdy nie zostaną zaszyfrowane nowym kluczem
- Konieczność przechowywania wielu starych kluczy

Rotacja kluczy - strategie

3. Bulk re-encryption:

- Odszyfrowanie wszystkich danych starym kluczem
- Zaszyfrowanie wszystkich danych nowym kluczem
- Jednorazowa operacja

Zalety:

- Natychmiastowe zakończenie użycia starego klucza
- Proste zarządzanie (tylko jeden aktywny klucz)

Wady:

- Wymaga przestoju lub znaczących zasobów obliczeniowych
- Ryzykowne (awaria w trakcie = utrata danych)
- Niemożliwe dla bardzo dużych zbiorów danych

Best practice: Envelope encryption + KEK rotation

- Zmień KEK, nie DEK

Rotacja kluczy - strategie

- DEK pozostają te same, ale są ponownie zaszyfrowane nowym KEK

6. Archiwizacja kluczy

Definicja archiwizacji:

- Długoterminowe przechowywanie kluczy, które nie są już aktywnie używane
- Klucze są nieaktywne, ale muszą być dostępne do odzyskania danych
- Różni się od backupu (backup = kopia aktywnych kluczy)

Kiedy archiwizować klucze?

TAK - archiwizuj:

- Klucze do deszyfrowania archiwalnych danych
- Klucze po zakończeniu rotacji (ale dane jeszcze istnieją)
- Klucze wymagane do compliance (np. podpisy umów)

NIE - zniszcz:

- Klucze do tymczasowej komunikacji
- Klucze sesyjne (TLS session keys)
- Klucze jednorazowe (ephemeral keys)
- Klucze skompromitowane

Cykl życia klucza kryptograficznego

- Klucze do weryfikacji historycznych podpisów

Archiwizacja kluczy - jak?

Wymagania dla archiwum kluczy:

1. Bezpieczeństwo:

- Szyfrowanie kluczy przed archiwizacją (Key Wrapping)
- Archiwum offline (air-gapped) lub w HSM
- Multi-factor authentication dla dostępu
- Fizyczne zabezpieczenia (sejf, skrytka bankowa)

2. Niezawodność:

- Redundancja: minimum 2-3 kopie w różnych lokalizacjach
- Okresowe testy odzyskiwania (disaster recovery drill)
- Ochrona przed degradacją nośników (migracja co 3-5 lat)

3. Dostępność:

- Udokumentowana procedura odzyskania
- Designated key custodians (osoby odpowiedzialne)

Archiwizacja kluczy - jak?

- SLA dla czasu odzyskiwania (np. 24h, 7 dni)

Archiwizacja kluczy - Key Recovery

Scenariusze odzyskiwania kluczy:

- Utrata klucza operacyjnego (awaria serwera, uszkodzenie HSM)
- Potrzeba odszyfrowania archiwalnych danych
- Audyt bezpieczeństwa / śledztwo
- Zmiana personelu (odejście administratora)
- Disaster recovery

Metody Key Recovery:

1. Secret Sharing (Shamir's Secret Sharing):

```
Klucz podzielony na N części (shares)
Odzyskanie wymaga K części (threshold)
Przykład: 5-of-3
- Klucz podzielony na 5 części
- Do odzyskania potrzeba 3 dowolnych części
- 2 części = brak możliwości odzyskania
```

Archiwizacja kluczy - Key Recovery

Zalety:

- Żadna pojedyncza osoba nie ma dostępu do klucza
- Odporność na utratę części kluczy
- Zgodność z “dual control” / “separation of duties”

Archiwizacja kluczy - Key Escrow

Key Escrow - powierzenie kopii klucza zaufanej trzeciej stronie

Modele Key Escrow:

1. Internal Escrow:
- Klucze przechowywane przez organizację
- Dostęp kontrolowany przez wewnętrzne procedury
- Używane w przedsiębiorstwach dla ciągłości biznesowej

2. Third-party Escrow:
- Klucze przechowywane przez zewnętrzną firmę (escrow agent)
- Dostęp tylko w określonych warunkach (umowa, nakaz sądowy)
- Używane w finansach, medycynie, administracji publicznej

3. Government Escrow (kontrowersyjne):
- Klucze dostępne dla organów ścigania
- Przykład: Clipper Chip (1993, USA) - porażka

Archiwizacja kluczy - Key Escrow

- Problemy: backdoor, zaufanie, kompromitacja

Ryzyka Key Escrow:

- Single point of failure
- Atrakcyjny cel dla atakujących
- Problemy prawne i etyczne
- Możliwość nadużyć

Best practice: Internal escrow + Secret Sharing

7. Niszczenie kluczy

Kiedy niszczymy klucze?

Klucz musi być zniszczony gdy:

- Zakończył się okres jego ważności (expired)
- Został skompromitowany lub podejrzenie kompromitacji
- Nie jest już potrzebny (np. zakończenie projektu, usunięcie użytkownika)
- Zmiana algorytmu kryptograficznego (migracja z RSA-2048 na RSA-4096)
- Wymagania compliance (PCI DSS, GDPR, HIPAA)

UWAGA: Niszczenie klucza ≠ usunięcie pliku!

Niszczenie kluczy - metody

Poziom 1 - Software (klucze w pamięci)

Problem: Zwykłe usunięcie zmiennej NIE niszczy klucza

```
# BŁĄD - klucz pozostaje w pamięci!
key = load_key()
# ... użycie klucza ...
del key  # To NIE wystarcza!
```

Niszczenie kluczy - metody

Właściwe podejście: Zerowanie pamięci - dlaczego to nie jest dobry przykład.

```
import ctypes
import secrets

# Bezpieczne zerowanie pamięci
def secure_zero(data):
    if isinstance(data, bytearray):
        ctypes.memset(id(data) + 32, 0, len(data))

# Użycie
key = bytearray(secrets.token_bytes(32))
# ... użycie klucza ...
secure_zero(key)  # Zerowanie przed usunięciem
del key
```

Niszczenie kluczy - metody

Właściwe podejście: Zerowanie pamięci (na poziomie niskopoziomowym)

```
# W praktyce w Pythonie brak jest przenośnego, gwarantowanego sposobu
# na wyzerowanie pamięci obiektu z kluczem.
# Produkcyjne systemy korzystają z prymitywów w C (np. OpenSSL, libsodium)
# lub z gotowych bibliotek kryptograficznych, które same dbają o lifecycle kluczy.
```

Niszczenie kluczy - metody

Dlaczego zwykłe usunięcie nie działa?

- Garbage collector w Pythonie/Java nie gwarantuje natychmiastowego usunięcia
- Klucz może pozostać w pamięci przez nieznany czas
- Może trafić do swap file na dysku
- Core dump może zapisać pamięć procesu
- Cold boot attack - pamięć RAM zachowuje dane przez kilka sekund po wyłączeniu

Zalecenia:

- Używaj `memset()` w C/C++ do zerowania pamięci
- W Pythonie: `cryptography.hazmat.primitives.constant_time` lub `ctypes.memset()`
- Wyłącz core dumps dla procesów obsługujących klucze: `ulimit -c 0`
- Szyfruj swap/pagefile (LUKS, BitLocker)

Niszczenie kluczy - metody

Poziom 2 - Pliki na dysku (SSD/HDD)

Problem: `rm` lub `delete` nie usuwa danych, tylko metadane!

HDD (magnetic disks):

- **Single overwrite** (DoD 5220.22-M): 1 pass z zerami/losowymi danymi
- **Multi-pass overwrite**:
  - DoD 5220.22-M: 3 passes (0xFF, 0x00, random)
  - Gutmann method: 35 passes (overkill dla nowoczesnych dysków)
- **Physical destruction**: degaussing (demagnetyzacja), niszczenie mechaniczne

```
# Bezpieczne usunięcie pliku (Linux)
shred -vfz -n 7 secret_key.pem
# -v: verbose, -f: force permissions, -z: final zero pass, -n: number of passes
# Niszczenie całego dysku
dd if=/dev/urandom of=/dev/sdX bs=1M
```

Niszczenie kluczy - metody

SSD/NVMe (Flash storage) - PROBLEM!

SSD używa **wear leveling** - dane są zapisywane w różnych miejscach fizycznych:

- Nadpisywanie tego samego pliku NIE gwarantuje nadpisania fizycznych sektorów
- Część danych może pozostać w nieużywanych komórkach flash

Niszczenie kluczy - metody

Właściwe podejście dla SSD:

1. Cryptographic Erasure (najlepsze dla SSD):

```
Zaszyfrowanie klucza kluczem głównym → zniszczenie klucza głównego
```

- Dane pozostają zaszyfrowane kluczem, który został zniszczony
- Niemożliwe odszyfrowanie bez klucza głównego
- Szybkie i skuteczne

2. ATA Secure Erase:

```
hdparm --security-erase PASS /dev/sdX
```

- Komenda sprzętowa dla SSD
- Kasuje wszystkie komórki pamięci flash
- Trwa 1-2 minuty dla całego dysku

Niszczenie kluczy - metody

3. Trim/Discard (dla pojedynczych plików):

```
# Usunięcie pliku + TRIM
rm secret_key.pem
fstrim -v /mount/point  # Informuje SSD, że sektor jest wolny
```

- SSD może rzeczywiście skasować dane podczas TRIM
- NIE gwarantowane natychmiastowo (controller decyduje)

Zalecenia dla SSD/NVMe:

- **Preferuj**: Full Disk Encryption (LUKS, BitLocker) + zniszczenie klucza szyfrującego
- **Nie polegaj na**: shred, dd, wielokrotne nadpisywanie
- **Dla krytycznych kluczy**: fizyczne zniszczenie dysku (kruszenie, topienie)

Niszczenie kluczy - HSM i sprzętowe

Hardware Security Modules

HSM oferuje dedykowane mechanizmy niszczenia kluczy:

1. Logiczne usunięcie:

```
# PKCS#11 API
C_DestroyObject(session, key_handle)
```

- HSM oznacza klucz jako usunięty
- Klucz staje się niedostępny dla aplikacji
- Fizyczna pamięć może być nadpisana później

2. Zeroizacja (fizyczne zerowanie):

```
# Komenda HSM-specific (przykład: Thales Luna)
lunacm> partition destroy -partition myPartition
```

- Natychmiastowe zerowanie pamięci flash/SRAM w HSM

Niszczenie kluczy - HSM i sprzętowe

- Nieodwracalne - klucze nie do odzyskania

Niszczenie kluczy - HSM i sprzętowe

3. Factory Reset:

- Przywrócenie HSM do stanu fabrycznego
- Kasuje WSZYSTKIE klucze i konfigurację
- Używane przy wycofywaniu HSM z eksploatacji

4. Fizyczne zniszczenie HSM:

- FIPS 140-2 Level 4: tamper-responsive - automatyczne kasowanie przy próbie włamania
- Wymagane dla wysoko klasyfikowanych danych
- Metody: shredding, incineration, chemical destruction

TPM (Trusted Platform Module):

- `tpm2_evictcontrol` - usunięcie klucza z NVRAM
- `tpm2_clear` - reset TPM (wymaga autoryzacji)
- Klucze efemeryczne (generowane w sesji) - automatycznie kasowane po reboot

Niszczenie kluczy - polityka i procedury

Dokumentacja procesu niszczenia

Organizacja musi mieć **Key Destruction Policy**:

1. Identyfikacja kluczy do zniszczenia:
- Monitoring dat wygaśnięcia
- Rejestr kluczy (key inventory)
- Automated alerts dla kluczy expired

2. Autoryzacja:
- Dual control: dwie osoby muszą zatwierdzić niszczenie
- Approval workflow (request → review → approve → execute)
- Audit trail: kto, kiedy, dlaczego zniszczył klucz

3. Weryfikacja zniszczenia:
- Potwierdzenie, że klucz został faktycznie zniszczony
- Test: próba użycia klucza (musi się nie udać)
- Dokumentacja w logach

Niszczenie kluczy - polityka i procedury

4. Dokumentacja i compliance:

```
Key Destruction Record:
- Key ID: 0x4A8F2B...
- Destruction date: 2025-01-15 14:32:18 UTC
- Method: AES Key Wrap deletion + cryptographic erasure
- Authorized by: John Doe (john.doe@example.com)
- Executed by: Jane Smith (jane.smith@example.com)
- Reason: Key rotation (scheduled)
- Verification: Attempted decryption failed (expected)
- Witness: Security Officer Mike Johnson
```

Compliance requirements:

- **PCI DSS**: klucze kart płatniczych muszą być zniszczone zgodnie z procedurą
- **GDPR**: "right to be forgotten" - klucze szyfrujące dane osobowe muszą być zniszczone
- **HIPAA**: klucze chroniące dane medyczne - udokumentowane niszczenie

Niszczenie kluczy - polityka i procedury

- NIST SP 800-88: wytyczne niszczenia mediów zawierających klucze

Niszczenie kluczy - najczęstsze błędy

Błędy przy niszczeniu kluczy

1. Backupy:

```
Problem: Klucz zniszczony w produkcji, ale pozostał w backup
```

- Niszczenie klucza musi obejmować WSZYSTKIE kopie
- Backupy, shadow copies, snapshots, repliki
- Klucze w systemach escrow

2. Logi i monitoring:

```
# BŁĄD - klucz w logu!
logger.info(f"Using key: {secret_key}")
```

- Klucze nie mogą trafić do logów
- Log files mogą być archiwizowane latami

Niszczenie kluczy - najczęstsze błędy

- Niszczenie klucza wymaga przeglądu logów

Niszczenie kluczy - najczęstsze błędy

3. Pośrednie kopie:

- Klucze w pamięci swap/pagefile
- Core dumps
- Hibernation file (Windows: hiberfil.sys)
- Virtual machine snapshots
- Container images (Docker layers)

4. Cloud storage:

```
Problem: Klucz usunięty z S3, ale pozostał w versioning
```

- AWS S3 versioning zachowuje stare wersje obiektów
- Soft delete w Azure/GCP - dane nie są natychmiast usunięte
- Repliki w innych regionach

Zalecenia:

- Inwentaryzacja wszystkich lokalizacji klucza
- Automated key destruction workflow
- Weryfikacja zniszczenia we wszystkich lokalizacjach
- Cryptographic erasure dla danych w chmurze (szyfruj klucze przed uploade)