

Sikkerhedstillæg: RAG Agent Chat Application

Dokument: Sikkerhedsanalyse og Forbedringsforslag

Version: 1.0

Dato: Januar 2026

Del A: Implementeret Sikkerhed

Oversigt over nuværende sikkerhedslag

Applikationen har implementeret følgende sikkerhedsmekanismer:

Sikkerhedslag	Status	Beskrivelse
HttpOnly Cookies	<input checked="" type="checkbox"/> Implementeret	Tokens utilgængelige for JavaScript
CSRF Protection	<input checked="" type="checkbox"/> Implementeret	Double Submit Cookie pattern
JWT Authentication	<input checked="" type="checkbox"/> Implementeret	Access + Refresh tokens
Azure AD SSO	<input checked="" type="checkbox"/> Implementeret	Enterprise single sign-on
Rate Limiting	<input checked="" type="checkbox"/> Implementeret	Per-endpoint begrænsninger
Internal Secret	<input checked="" type="checkbox"/> Implementeret	NGINX-til-backend verifikation
Audit Logging	<input checked="" type="checkbox"/> Implementeret	Request/response logging
Input Validering	<input checked="" type="checkbox"/> Implementeret	Pydantic + token limits
CORS	<input checked="" type="checkbox"/> Implementeret	Strict origin control
XSS Sanitering	<input checked="" type="checkbox"/> Implementeret	DOMPurify i frontend

A.1 HttpOnly Cookies

Hvad det beskytter mod: XSS token-tyveri

Implementation:

```
response.set_cookie(  
    key="access_token",  
    value=token,  
    httponly=True,      # JavaScript kan IKKE læse  
    secure=True,        # Kun HTTPS  
    samesite="lax"       # CSRF beskyttelse  
)
```

Styrke: Selv ved succesfuldt XSS-angreb kan angriber ikke stjæle tokens.

A.2 CSRF Protection

Hvad det beskytter mod: Cross-Site Request Forgery

Implementation: Double Submit Cookie pattern med signeret token

Flow: 1. Server genererer CSRF token ved login 2. Token gemmes i læsbar cookie (ikke HttpOnly) 3. Frontend sender token i X-CSRF-Token header 4. Backend verificerer cookie = header

Styrke: Angriber kan ikke læse cookie fra andet domæne (Same-Origin Policy).

A.3 JWT Authentication

Hvad det beskytter mod: Uautoriseret adgang

Implementation: - Access token: 60 minutter levetid - Refresh token: 7 dage levetid - HS256 signing med min. 32 tegn secret

Styrke: Stateless authentication med automatisk udløb.

A.4 Rate Limiting

Hvad det beskytter mod: Brute force, DDoS, API misbrug

Implementation: | Endpoint | Grænse | |-----|----| | /auth/* | 5/minut | | /chat | 10/minut | | /prompts | 20/minut |

Styrke: Begrænsrer automatiserede angreb.

A.5 Audit Logging

Hvad det giver: Sporbarhed og compliance

Logger: - Bruger ID, IP, timestamp - Endpoint og metode - Saniteret request body - Response status og tid - Token forbrug

Styrke: Mulighed for at opdage og efterforske sikkerhedshændelser.

Del B: Anbefalede Forbedringer

B.1 KRITISKE Forbedringer (Høj prioritet)

1. Secrets Management

Problem: API keys og secrets ligger i .env fil og kan komme i version control.

Løsning:

```
# Brug Docker secrets eller vault
services:
  backend:
    secrets:
      - openai_api_key
      - jwt_secret

secrets:
  openai_api_key:
    external: true # Fra Docker Swarm eller Kubernetes
```

Alternativ: HashiCorp Vault, AWS Secrets Manager, Azure Key Vault

2. Password Hashing Opgradering

Problem: Koden bruger bcrypt, men bør verificere work factor.

Løsning:

```
from passlib.context import CryptContext

pwd_context = CryptContext(
    schemes=["argon2", "bcrypt"], # Argon2 er nyere og stærkere
    deprecated="auto",
    argon2_memory_cost=65536, # 64 MB
    argon2_time_cost=3, # 3 iterationer
    argon2_parallelism=4 # 4 tråde
)
```

3. Token Rotation ved Refresh

Problem: Refresh token forbliver det samme indtil udløb.

Løsning: Implementer token rotation

```

async def refresh_access_token(self, refresh_token: str):
    # Verificer refresh token
    token_data = self.verify_token(refresh_token, "refresh")

    # Generer NYT refresh token (rotation)
    new_refresh_token = self.create_refresh_token(user)

    # Invalidere det gamle (kræver token blacklist)
    await self.blacklist_token(refresh_token)

    return {
        "access_token": new_access_token,
        "refresh_token": new_refresh_token # NYT token
    }

```

4. Token Blacklist/Revocation

Problem: Logout invaliderer ikke tokens - de virker til udløb.

Løsning: Redis-baseret token blacklist

```

import redis

class TokenBlacklist:
    def __init__(self):
        self.redis = redis.Redis(host='redis', port=6379)

    async def blacklist(self, token: str, exp: datetime):
        ttl = (exp - datetime.utcnow()).seconds
        await self.redis.setex(f"blacklist:{token}", ttl, "1")

    async def is_blacklisted(self, token: str) -> bool:
        return await self.redis.exists(f"blacklist:{token}")

```

B.2 VIGTIGE Forbedringer (Medium prioritet)

5. Content Security Policy (CSP)

Problem: Ingen CSP header beskytter mod XSS.

Løsning: Tilføj CSP middleware

```

@app.middleware("http")
async def add_security_headers(request: Request, call_next):
    response = await call_next(request)

```

```
response.headers["Content-Security-Policy"] = (
    "default-src 'self'; "
    "script-src 'self' https://cdn.jsdelivr.net https://cdnjs.cloudflare.com;
"
    "style-src 'self' 'unsafe-inline' https://cdnjs.cloudflare.com; "
    "img-src 'self' data:; "
    "connect-src 'self' https://api.openai.com; "
    "frame-ancestors 'none'; "
    "base-uri 'self';"
)
response.headers["X-Content-Type-Options"] = "nosniff"
response.headers["X-Frame-Options"] = "DENY"
response.headers["Referrer-Policy"] = "strict-origin-when-cross-origin"
return response
```

6. Request ID Tracking

Problem: Svært at korrelere logs på tværs af services.

Løsning:

```
import uuid

@app.middleware("http")
async def add_request_id(request: Request, call_next):
    request_id = request.headers.get("X-Request-ID", str(uuid.uuid4()))

    # Tilføj til request state
    request.state.request_id = request_id

    response = await call_next(request)
    response.headers["X-Request-ID"] = request_id

    return response
```

7. Structured Logging (JSON)

Problem: Tekstbaserede logs er svære at parse og analysere.

Løsning:

```
import structlog

structlog.configure(
    processors=[
```

```

        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.JSONRenderer()
    ]
)

logger = structlog.get_logger()

# Brug:
logger.info("chat_request",
            user_id=user.id,
            session_id=session_id,
            tokens_used=tokens,
            response_time_ms=elapsed
)

```

Output:

```
{"timestamp": "2026-01-31T10:00:00Z", "event": "chat_request", "user_id": "123",
"tokens_used": 450}
```

8. IP-baseret Account Lockout

Problem: Rate limiting er per-endpoint, ikke per-bruger/IP kombination.

Løsning:

```

class AccountLockout:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.max_attempts = 5
        self.lockout_minutes = 15

    async def record_failure(self, username: str, ip: str):
        key = f"login_fail:{username}:{ip}"
        attempts = await self.redis.incr(key)
        await self.redis.expire(key, self.lockout_minutes * 60)

        if attempts >= self.max_attempts:
            await self.lock_account(username, ip)

    async def is_locked(self, username: str, ip: str) -> bool:
        return await self.redis.exists(f"locked:{username}:{ip}")

```

B.3 ANBEFALEDE Forbedringer (Lav prioritet)

9. API Versioning

Problem: Breaking changes vil påvirke alle klienter.

Løsning:

```
from fastapi import APIRouter

v1_router = APIRouter(prefix="/api/v1")
v2_router = APIRouter(prefix="/api/v2")

app.include_router(v1_router)
app.include_router(v2_router)
```

10. Health Check Forbedring

Problem: /health tjekker kun at API'et kører.

Løsning: Deep health check

```
@router.get("/health")
async def health_check():
    checks = {
        "api": "healthy",
        "database": await check_chromadb(),
        "openai": await check_openai_api(),
        "redis": await check_redis() if REDIS_ENABLED else "disabled"
    }

    all_healthy = all(v == "healthy" or v == "disabled" for v in checks.values())

    return {
        "status": "healthy" if all_healthy else "degraded",
        "checks": checks,
        "timestamp": datetime.utcnow().isoformat()
    }
```

11. Secrets Rotation Strategi

Problem: Secrets roteres sjældent eller aldrig.

Anbefaling: | Secret | Rotationsfrekvens | |-----|-----| | JWT_SECRET_KEY | Hver 90 dage | | API_SECRET_KEY | Hver 90 dage | | OPENAI_API_KEY | Ved kompromittering | | Database passwords | Hver 90 dage |

Tip: Understøt multiple JWT secrets under rotation:

```
JWT_SECRET_KEYS = [
    settings.jwt_secret_key,          # Aktuel
    settings.jwt_secret_key_old      # Forrige (til verifikation)
]
```

12. Dependency Scanning

Problem: Sårbare dependencies opdages ikke automatisk.

Løsning: Tilføj til CI/CD pipeline

```
# GitHub Actions eksempel
- name: Security scan
  run:
    - pip install safety pip-audit
      safety check
      pip-audit
```

Del C: Prioriteret Handlingsplan

Fase 1: Kritisk (Inden produktion)

1. Implementer secrets management (Vault/Docker secrets)
2. Tilføj token blacklist med Redis
3. Implementer token rotation ved refresh
4. Tilføj security headers (CSP, X-Frame-Options, etc.)

Fase 2: Vigtig (Første måned)

5. Opgrader til Argon2 password hashing
6. Implementer account lockout
7. Tilføj request ID tracking
8. Skift til structured JSON logging

Fase 3: Forbedring (Løbende)

9. API versioning

10. Deep health checks
 11. Automatisk dependency scanning
 12. Secrets rotation procedure
-

Del D: Opsummering

Nuværende sikkerhedsniveau: GODT

Applikationen har implementeret de vigtigste sikkerhedsmekanismer og er væsentligt bedre sikret end mange lignende løsninger. HttpOnly cookies, CSRF protection og rate limiting giver solid beskyttelse mod de mest almindelige angrebstyper.

Efter forbedringer: FREMRAGENDE

Med de foreslæde forbedringer vil applikationen have enterprise-grade sikkerhed med:

- Komplet token lifecycle management
- Defense in depth på alle lag
- Fuld sporbarhed og compliance-readiness
- Automatiseret sikkerhedsovervågning

Dokument slut

Udarbejdet: Januar 2026