. Säkerhetsarkitektur och dokumentation

JWT vs sessions: fördelar, nackdelar och när man använder vad

I min projekt har jag valt jwt då applikationen är REST-baserad, stateless. Jag hanterar säkerhet genom att validera token samt använda token black list för att ogiltigförklara token vid utloggning. Fördelarna är att kräver det inte serverside-session lagring , stateless authentication system är enklare att förstå. Lämplig att använda i REST APIs. Klienten bär med sig användarinformation i token. Används också för skalbarhet i programmet.

Nackdelar:

- Ingen enkel mekanism för att ogiltigförklara token direkt,löses med token-blacklist i vårt projekt. Med blacklist servern måste komma ihåg de blacklistade token vilken motverkar fördelar med att servern inte behöver spara data.
- Sessionsbaserad autentisering är enklare att implementera för sessions invalidation på serversidan. Klienten får en session-Id i form av en cookie som servern använder för att identifiera user och hämta dess session. Det är enkelt att ta bort session från servern och nvänds ofta vid applikationer med behov av omedelbar sessionshantering och kontroll.
- **Nackdelar:**

Sessions orsaker skalbarhetsproblem eftersom att sessioner lagras på servern kan vara krångligt att använda för webbapplikationen med flera domän.

Threat model – Hot vi skyddar mot och Säkerhetsarkitektur och dokumentation

Applikationen skyddar sig mot flera vanliga säkerhetshot genom en kombination av tekniker och ramverk funktioner. Applikationen är byggd med ett starkt fokus på säkerhet enligt principer för modern webbsäkerhet. Nedan förklaras de viktigaste säkerhetsåtgärderna, designbeslut, hot modell och risker.
Det används JWT Blacklist för att motverka token replay-attacker eller förhindrar att en token blivit stulen och vid utloggning säkerställs att den inte längre kan användas, även om den fortfarande är giltig till en viss tid.
När det gäller CSRF (Cross-Site Request Forgery), minskar riskerna kraftigt eftersom applikationen är stateless och använder JWT i headern istället för cookies. Eftersom inga sessions-cookies används i klienten, blir CSRF-attacker i praktiken inte möjliga i denna arkitektur.

Även om den här applikationen inte har HTML-sidor, och inte körs i en webbläsare direkt, bara används via Postman har jag ändå lagt till säkra HTTP-rubriker i config-fil för säkerhett om

någon klient någonsin använder API:et via en webbläsare eller frontend. Den ger skydd mot XSS och Clickjacking. X-Frame-Options stoppar andra webbplatser från att visa att någon försöker lura användare att klicka fel.Content-Security-Policy bestämmer content användnig, så att skadlig kod inte kan köras.

För att hindra obehörig åtkomst, används rollbaserad åtkomstkontroll i Spring Security. Endast användare med korrekt roll, som USER eller ADMIN, får åtkomst till specifika endpoints. Detta styrs med hjälp av @PreAuthorize och hasRole i kontroller .
Det finns ett skydd mot API-missbruk genom att använda Bucket4j in-memory rate limiting. Det begränsar antalet anrop per IP-adress och endpoint inom ett visst tidsintervall.

Projektet har konfigurerat HTTP Strict Transport Security (HSTS) för att förhindra att trafiken sänds över osäker HTTP. Detta skyddar användare vid framtida besök.
För att hindra obehörig åtkomst, används rollbaserad åtkomstkontroll i Spring Security. Endast användare med korrekt roll, som USER eller ADMIN, får åtkomst till specifika endpoints. Detta styrs med hjälp av @PreAuthorize och hasRole()-kontroller i konfigurationen.

Förutom validering används en Input Sanitizer för att rensa användarinmatning från skadlig HTML eller skript. Sanitizern används till exempel för boktitlar och användares namn som kommer från klienten via @RequestBody.

Avslutningsvis använder jag en kortlivad access-token för autentisering,minskar risk vid stöld och en längre refresh-token för att begära nya tokens som används mer sällan.

Kvarvarande säkerhetsrisker och begränsningar

Trots att applikationen har ett starkt säkerhetsskydd finns det några kvarstående risker och begränsningar
JWT är fortfarande giltig tills den går ut, om den blir stulen. Token kan endast ogiltigförklaras via blacklist. HTTPS ännu inte aktiverat, vilket gör trafiken oskyddad mot avlyssning. För fullständig skydd mot attacker krävs att applikationen körs över HTTPS i produktion,
Slutligen används en in-memory lösning för rate limiting med hjälp av Bucket4j. Detta fungerar perfekt i ett enskilt system, Men om man har flera servrar (t.ex. i ett större system) så är det bättre att spara den informationen centralt, till exempel i Redis.
Följande är också inte implementerad med store krav på säkerhet

- Implementera två-faktor autentisering (2FA) med TOTP
- "Remember me" funktionalitet med säkra persistent tokens
- Lösenordsåterställning via e-post med säkra tokens
- Genomför penetration testing av er egen applikation
- Använd säkerhetsverktyg som OWASP ZAP
- Implementera audit logging av alla kritiska säkerhetshändelser
- Automatisk varning vid misstänkta säkerhetsincidenter

Säkerhet kontra användarupplevelse

Applicationen verkar ha en balans mellan säkerhet och att det ska vara lätt att använda systemet. Till exempel används det JWT för att göra systemet snabbt och smidigt och Rate Limiting som stoppar möjliga missbruk men vanliga användare märker inte det eftersom en användare gör oftast bara några få anrop åt gången samt finns det tydliga meddelande för användare till exampel hur lösenord ska vara eller vad saknas i olika fälten.
Alla säkerhetsfunktioner sker i bakgrunden, så användaren behöver inte tänka på dem.