OAuth versioner 1.0a och 2.0

- En säkerhetsjämförelse

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH Ingenjörshögskolan vid Campus Helsingborg
Institutionen för Elektro- och informationsteknik

Examensarbete:
Joel Lindholm
Johan Thorsberg

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

Tryckt i Sverige
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2013

Sammanfattning

I detta arbete har två versioner av OAuth analyserats, protokollet OAuth 1.0a och det senare ramverket OAuth 2.0. Senare versionsnummer anses ofta vara bättre, men OAuth 2.0 har mött mycket kritik. Det har fått kritik för att inte vara lika säkert som OAuth 1.0a. Samtidigt menas det att OAuth 1.0a är ett mycket krångligt protokoll att implementera, vilket har stoppat det från att växa som det var tänkt. Utöver detta analyseras även LinkedIns API-lösningar av de båda versionerna.

Här följer en kort förklaring på vad OAuth löser för problem: En resursägare har resurser på en server. En tredjepart vill ta del av dessa resurser i resursägarens namn. OAuth löser detta genom att resursägaren autentiserar sig mot servern och godkänner att tredjepart har befogenhet att komma åt de resurser ägaren besitter.

Resultat av analysen ledde till att OAuth 2.0 valdes till implementering av en OAuth-klient till LinkedIn på PMCG Scandinavia ABs projektportal. Resultatet är en OAuth 2.0 lösning som ger LinkedInanvändare möjligheten att logga in på projektportalen genom LinkedIn. LinkedIns OAuth 2.0 lösning ansågs vara tillräckligt säker och mycket enklare att implementera och underhålla.

Nyckelord: OAuth, nätverkskommunikation, autentisering, befogenhet, protokoll, ramverk

Abstract

In this work two versions of Oauth have been analyzed, the protocol OAuth 1.0a and the newer framework OAuth 2.0. A higher version number is often considered a good thing, but OAuth 2.0 has encountered much criticism. It has been criticised of not being safe enough while OAuth 1.0a was criticised of being very complicated protocol to implement, which has stopped it from growing as expected.

The following problem is solved with OAuth: a resource owner has resources on a server. A third party would like to use some of these resources in the resource owner's name. OAuth solves this by letting the resource owner authenticates at the server and agree that the third party is authorized to access the resources that the resource owner possesses.

The result of this work led to an implementation of an OAuth client to LinkedIn on PMCG Scandinavia AB's project portal. The result is an OAuth 2.0 solution that gives LinkedIn users the ability to log in to the project portal through LinkedIn. LinkedIns OAuth 2.0 solution was considered to be sufficiently safe and much easier to implement and maintain.

Keywords: OAuth, network communication, authentication, authorization, protocol, framework

Innehållsförteckning

1 Inledning

1.1 Bakgrund

PMCG Scandinavia AB är ett företag med projektledare och konsulter inom IT Service Management. Deras mål är att implementera processer och verktyg som hjälper och förbättrar företags affärsverksamhet. Detta gör de genom att fokusera på att optimera utnyttjandet av kundens IT-resurser.

PMCG har en egen projektportal, där dess anställda och kunder kan logga in för att bland annat statusuppdatera sina nuvarande projekt. Idag måste man skapa en användare till projektportalen för att kunna ta del av informationen. Detta skulle PMCG vilja ändra på, genom att låta nya och gamla användare logga in med sina LinkedIn-konton.

LinkedIn är ett socialt nätverk, med syfte att varje användare enkelt skall kunna knyta nya kontakter och upprätthålla gamla. Till skillnad från det sociala nätverket Facebook har LinkedIn inriktat sig mer på professionellt bruk, och erbjuder sina användare verktyg att beskriva sina kunskaper och arbetserfarenhet. LinkedIn erbjuder tjänsten att loggning via LinkedIn med hjälp av OAuth.

Ett exempel på vad som möjliggörs med OAuth. En resursägare har resurser på en server. En tredjepart vill ta del av dessa resurser i resursägarens namn. Detta går till så att resursägaren autentiserar sig mot servern och godkänner att tredjepart har befogenhet att komma åt de resurser ägaren besitter.

Protokollet OAuth 1.0 publicerades 2007 och blev senare uppdaterat 2009 till OAuth 1.0a. Detta eftersom en säkerhetsbrist upptäckts i specifikationen. Oktober 2012 släpptes OAuth 2.0 till allmänheten, dock nu som ett ramverk. Syftet med en ny version var att lösning i OAuth 1.0a anses svår att implementera[4,5]. OAuth 2.0 har dock mött mycket kritik för att vara mindre säker än sin föregångare[5]. Nu måste utvecklarna av ett OAuth API implementera mycket extra säkerhet själva, om de vill uppnå samma säkerhetsnivå som i OAuth 1.0a. Detta kräver mycket kunskaper kring it- och kommunikationssäkerhet, vilket inte alla besitter. Det leder också till att olika servrar har olika lösningar, som i sin tur leder till extra arbete för de som vill koppla upp sig mot fler än ett API.

1.2 Syfte och målsättning

Syftet med examensarbetet är att analysera skillnaderna mellan OAuth 1.0a och OAuth 2.0. Tre områden kommer att undersökas. Hur dokumentationen

ser ut, hur mycket arbete krävs för implementation samt hur säkra de är gentemot varandra.

Målet är att PMCG Scandinavia ABs anställda och kunder skall kunna säkert logga in på sin projektportal med sina LinkedInkonton, genom en implementation av OAuth.

Exempel på scenario som vill uppnås:
LinkedIns server har information om sina användare och har skapat ett OAuth API. Detta API låter en tredjepart, i detta fall PMCGs projektserver, att komma åt användares information, men endast med användarens godkännande. När projektservern vill komma åt en användares uppgifter leds användaren till LinkedIn. Där får användaren logga in och godkänna PMCGs tillgång till dess information. Användaren skickas sedan tillbaka till projektservern. Nu kan projektportalen hämta information som användaren har tillgång till. Informationen kan vara exempelvis namn, email eller bilder. Utöver funktionen att komma åt data på andra serverar så slipper användaren dela sitt lösenord och användarnamn med PMCG.

1.3 Problemformulering

Grundfrågan för rapporten är:
Vilken OAuth version skall användas vid implementationen hos PMCGs projektportal.

Delfrågor:

- Hur har de två versionerna av OAuth implementerats i LinkedIns servrar?
- Vilken av de två versionerna av OAuth kan inom rimlig tid implementeras till PMCGs servrar?
- Vilken säkerhetsnivå ligger de två versionerna av OAuth på?
- Hur ser framtiden ut för de två versionerna av OAuth?

1.4 Avgränsningar

Analysen kommer endast involvera OAuth, version 1.0a och 2.0, andra protokoll och ramverk som kan tänkas användas till en liknande lösning behandlas inte i denna rapport.

I OAuth 2.0 specificeras ett antal autentiseringsflöden. LinkedIn har implementerat endast ett av dessa, vilket är *Authorization Code Grant*, därmed kommer enbart detta flöde att analyseras mer ingående.

Lösningen kommer endast involvera hemsidorna LinkedIn och PMCGs projektportal.

PMCGs projektportal är uppbyggd i Joomla 1.5, som är ett *content management system*. Dessa system används för att smidigt hantera större applikationer utan krav på programmeringskunskaper, exempelvis en hemsida. I Joomla används språken PHP och Javascript samt MySQL-databaser, alltså är programmeringsspråken och struktur redan valda.

2 Teknisk bakgrund

2.1 Terminologi

För att förstå OAuth, så är det viktigt att känna till en del termer som förklaras nedan. Att hitta direkta översättningar mellan svenska och engelska ord inom data- och IT-världen kan vara svårt då dessa inte alltid finns. Att då behålla de engelska termerna istället för att hitta på nya ord är att föredra. Detta medför att en del engelska termer kommer användas i rapporten.

2.1.1 OAuth-relaterade termer

API (*Application programming interface*)

> Ett API är ett protokoll skapat för att möjliggöra att olika typer av mjukvara skall kunna kommunicera med varandra.

Autentisering (*Authentication*)

> Att bevisa att personen/klienten i fråga verkligen är den som den uppger sig att vara.

Befogenhet (*Authorization*)

> Att ge någon/något möjligheten att utföra en eller flera handlingar i någon annans namn. OBS: Det engelska ordet, authorization, skall inte blandas ihop med authentication.

Server

> I denna rapport: En http-server med ett OAuth-API som kan ta emot och svara på OAuthförfrågningar.

Klient (*Client*)

> I denna rapport: Den som ställer förfrågningar till en servers OAuth-API i resursägarens namn. Exempelvis när PMCGs projektportal skickar förfrågningar till LinkedIn, angående användarinformation.

Skyddade resurser (*Protected resources*)

> Resurser som ägs av resursägaren(se nästa term). Resursägaren kan exempelvis vara en användare hos LinkedIn och de skyddade resurserna kan då vara användarnamn och emailadress.

Resursägare (*Resource owner*)

Den som äger och kontrollerar sina skyddade resurser. Ett exempel på en resursägare är en användare på LinkedIn.

Credentials

Unika identifierare, exempelvis användarnamn och lösenord.

OAuth 1.0a definierar tre olika typer av *credentials*:

- *Client*, används för att identifiera och autentisera klienten som ställer förfrågan.
- *Temporary*, används vid en autentiseringsförfrågan.
- *Token*, används vid förfrågan om resurser.

OAuth 2.0 definierar två olika typer av *credentials*:

- *Client*, används för att identifiera och autentisera klienten som ställer förfrågan.
- Vid flödet *Resource owner password credential grant* används resursägarens lösenord och användarnamn som *credentials*.

Token/Access token (OAuth 1.0a respektive 2.0)

En unik identifierare utdelad till klienten av servern. Används av klienten för att visa den har blivit godkänd av resursägaren, och kan ställa förfrågningar med resursägarens befogenhet. Dessa *tokens* har en livstid som är specificerad av servern, livstiden kan vara oändlig.

I OAuth 1.0a måste klienten kunna bekräfta sitt ägande av en *token*, och sin befogenhet att representera resursägaren. För detta används en privat/delad hemlighet. Något liknande finns inte specificerat i OAuth 2.0.

Refresh token

En unik identifierare utdelad till klienten av servern, i OAuth 2.0. En *refresh token* används när en *access tokens* livstid passerats. Klienten skickar då en förfrågan om att bli tilldelad en ny med hjälp av en *refresh token*. En *refresh token* har liksom *access tokens* en specificerad livstid.

Bearer token

En *bearer token* kräver ingen autentisering av klienten eller resursägaren som ställer förfrågningar. Exempelvis kan en *access token* användas som en *bearer token*. Detta är standardtypen av *tokens* i OAuth 2.0.

Resursserver (*Resource server*)

Servern som håller i de skyddade resurserna, exempelvis en av LinkedIns servrar. Den kan acceptera och svara på förfrågningar angående de skyddade resurserna.

Befogenhetsserver (*Authorization server*)

Server som kontrollerar autentisering av klienter och resursägare, samt delar ut *access tokens* och eventuellt *refresh tokens*. Exempelvis kan en av LinkedIns servrar agera befogenhetsserver

URI (*Uniform resource identifier*)

URI är en textsträng som i OAuth används för att identifiera ett namn eller en webbresurs. URIs följer fördefinierade regler om hur de skall ser ut. En webbadress(URL) är ett exempel på en typ av URI.

Scope

Ett scope är en sträng som specificerar vilka skyddade resurser som klienten vill ha tillgång till.

State

En slumpmässigt genererad textsträng, som skall vara omöjlig att gissa. State parametern ger skydd mot så kallade *Cross-site forgery request* (CSFR). State används för att säkra att det är samma resursägare som ställde autentiseringsförfrågan och den som faktiskt autentiserade sig. Detta sker genom att den måste var identisk vid båda förfrågningarna, samt att den har en begränsad livstid.

Svarstyp (*Response type*)

Används endast i OAuth 2.0. Definiera vilken typ av svar klienten förväntar sig från befogenhetsservern, när resursägaren är autentiserad.

2.1.2 Attacker

Cross-site forgery request (CSFR)

Utnyttjar förtroendet som servern har för sina användare. En CSRF attack kan ske genom att en inkräktare har skapat en länk eller ett script som utför en viss handling på en hemsida, där en användare är känd eller rent av autentiserad. Detta ger sken av att det är användaren som utför handlingen.

Cross-site scripting (XSS)

Utnyttjar förtroendet som användaren har för en hemsida. Hemsidan visas fast med angriparens gömda kod i bakgrunden, för att på så sätt stjäla information.

Clickjacking

Att lura användaren att trycka på en osynlig knapp, där angriparen placerat sin egen kod. Knappen kan exempelvis placeras ovanpå en annan betrodd knapp.

Eavesdropping

Avlyssnar trafik mellan två punkter på ett nätverk i hemlighet.

Man-in-the-middle(MITM)

Attack när angriparen placerar sig själv mellan två noder i ett nätverk. När den ena noden skickar data hamnar denna hos angriparen som själv skickar detta vidare till den egentliga mottagaren. Detta ger sken av att de två noderna talar med varandra, och att all data som tas emot inte har förändrats.

2.2 SSL/TLS

Secure Socket Layer (SSL) och det nya Transport Layer Security (TLS) är krypteringsprotokoll och används mellan två punkter i ett nätverk. Målet är att data som skickas mellan punkterna inte ska vara begriplig för andra än sändaren och mottagaren. Dessutom autentiseras de inblandade med hjälp av certifikat, om så önskas. Användningen av SSL/TLS idag är utbredd, exempelvis används den av banker och email-tjänster. Säkerheten anses vara mycket hög för tillfället. Att försöka attackera en SSL/TLS kommunikation som redan är upprättad betraktas som mycket svårt. Inkräktare väljer istället för att ge sig på redan upprättad kommunikation genom att tränga sig in när

kommunikationen skall sättas upp, med bland annat man-in-the-middle(MITM).

Båda versionerna av OAuth använder sig helt eller dels av SSL/TLS. I OAuth 1.0a används det för att säkert skicka nycklar mellan server och klient i autentiseringsflödet. OAuth 2.0 använder sig av SSL/TLS för att både skicka data säkert och för autentisering av klienten.

2.3 Om OAuth 1.0a

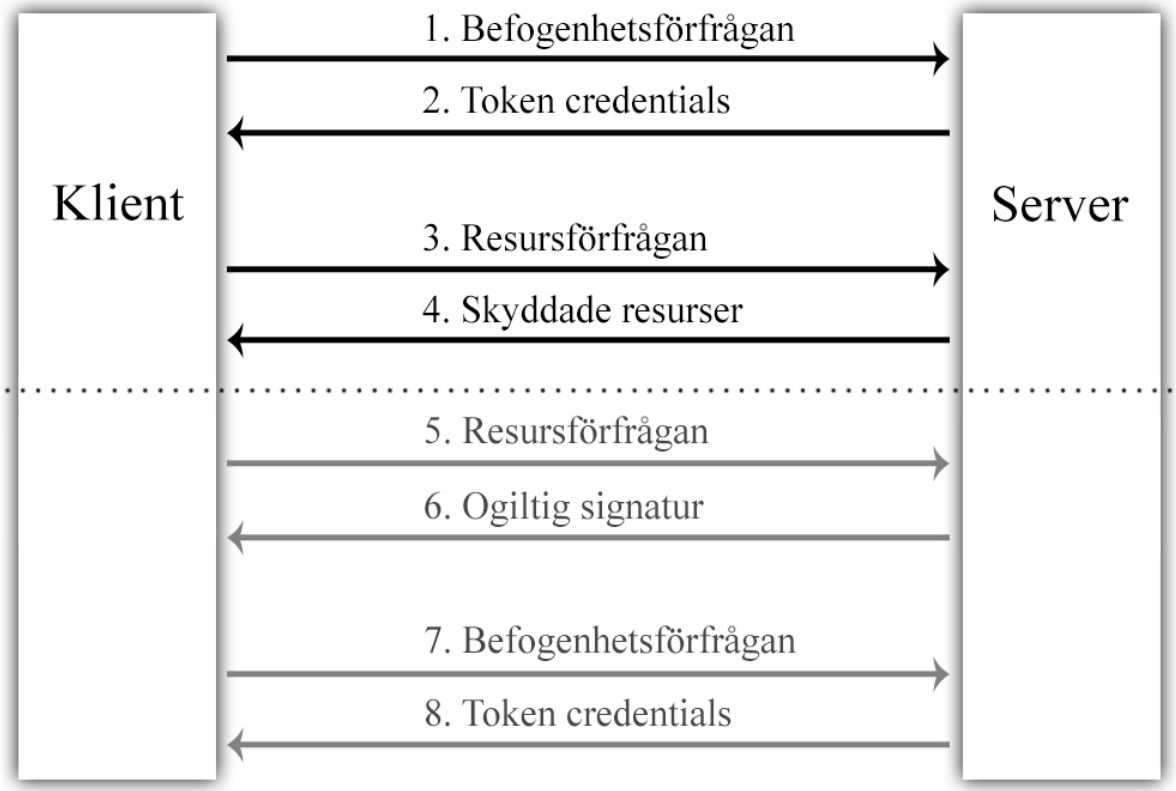
Detta delkapitel går igenom hur resursägaren, klienten och servern bär sig åt för att kommunicerar med varandra i protokollet OAuth 1.0a.

2.3.1 Signaturer

OAuth 1.0a använder signaturer som ett verktyg för servern att kunna kontrollera om ett meddelandes ursprung är genuint. Rapporten kommer inte gå igenom exakt hur en signatur skall byggas upp, utan hänvisar till specifikationen[1] eller andra betrodda källor.

Signaturerna i OAuth 1.0a bygger på att klienten och servern har nycklar som endast de känner till, så kallade *credentials*. För att skicka en resursförfrågan måste en signatur skickas med. Signaturen byggs upp med hjälp av klientens c*lient-* och *token credentials* samt de andra parametrarna i förfrågan. Detta leder till att varje enskild förfrågan har en unik signatur som inte kan förstås av någon som inte har tillgång till nycklarna. Med användningen av tidstämplar och *nounce* i signaturen kan varje unik signatur endast användas en gång. När servern tar emot en förfrågan jämförs dess signatur med en egengenererad signatur, med hjälp av sina egna nycklar. Om de två signaturerna är likadana så kan servern dra slutsatsen att klienten i fråga verkligen är den som skickade förfrågan.

2.3.2 OAuth 1.0a överblick



Figur 1: OAuth 1.0a flödesschema

OAuth 1.0a används för att ge en tredje part, klienten, tillgång till resursägarens skyddade resurser. Figur 1 ovan ger en överblick över hur det fungerar.

1. Först ber klienten servern att få autentisera sig och resursägaren.

2. Om steget innan går bra får klienten ett antal *token credentials*. Nu har klienten möjligheten att komma åt de skyddade resurser som resursägaren har godkänt.

3. De skyddade resurserna kommer klienten åt genom att fråga servern. Med sin förfrågan skickas en signatur, som har skapats med hjälp av klientens *token credentials*.

4. Om allt står rätt till kommer servern svara med de skyddade resurserna som klienten har frågat efter.

5-8. Beskriver vad som kan hända om servern har definierat en livstid på *token credentials*. Klienten ställer en resursförfrågan och får nej som

svar från servern. Klienten måste då återautentisera sig och sin användare och bli tilldelad nya *token credentials* enligt pilarna 1 - 2.

2.3.3 Förberedelser

För att en klient skall kunna erbjuda en resursägare några OAuthtjänster, måste den registrera sig hos en server. När en sådan registrering sker blir klienten tilldelad klient-credentials: klientidentifierare och en privat/delad hemlighet. Hemligheten används för att skapa signaturer till klientens förfrågningar. Servern kan be klienten registrera i förväg vart svar skall skickas, annars måste en svars-URL skickas med i varje fråga. Utöver detta måste även klienten konfigureras så att den använder serverns tre förberedda ändpunkter (endpoints), så den vet var den skall skicka sina förfrågningar:

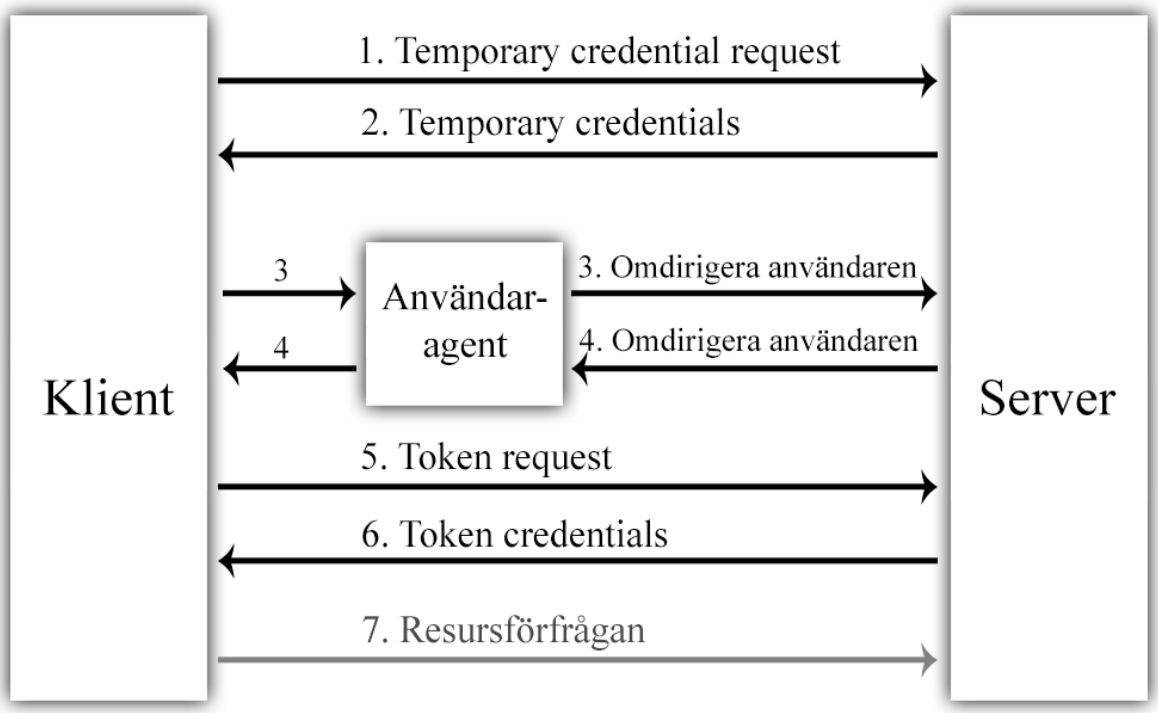
- *Temporary credential request*
- *Resource owner authorization*
- *Token request*

2.3.4 Autentiseringsflöde

I detta avsnitt beskrivs hur klienten får tag på ett par *token credentials*. Flödet som beskrivs nedan är en djupdykning av steg 1-2 i figur 1 ovan.

Utöver de parametrarna, från klienten till server, som nämns nedan så skickas även dessa alltid med:

- Consumer key
  klientidentifieraren som klienten fått vid registrering hos servern.
- Signature metod
  Vilken metod som signaturen har tagits fram genom.
- Timestamp
  Tidstämpel, sekunder sedan 1 januari 1970 00:00:00.
- Nonce
  En slumpmässig genererad sträng av klienten. Servern använder denna tillsammans med tidsstämpeln för att se om förfrågan har gjorts förut. Ett *nounce* får endast användas en gång.



Figur 2: Autentiseringsflöde för OAuth 1.0a

När resursägaren vill ha sina resurser på servern genom klienten måste alla steg i Figur 2 följas, om klienten inte tidigare har blivit godkänt av resursägaren.

1. När klienten skall hämta data från servern måste den först autentisera sig själv. Detta gör den med en förfrågan till den första ändpunkten, *Temporary credential request*. Utöver de andra parametrarna skickas en signatur, baserad på klienthemligheten, och eventuellt en callback. Denna parameter berättar för servern var den skall skicka sitt svar. Svaret i nästa steg skickas i klartext, servern skall alltså begära att meddelanden i steg 1 och 2 skickas med säkra medel, exempelvis SSL/TLS.

2. Servern har nu tagit emot klientens förfrågan. Den kontrollerar så att alla parametrar finns med och att signaturen stämmer. Om allt är som det skall så skapar den två *temporary credentials*, en identifierare och en hemlighet. De fungerar som en sorts användarnamn och lösenord för klienten under detta flöde. Dessa skickas tillbaka till adressen som den fick med i förfrågningen, eller till en svars-URL som registrerats i förväg.

3. Nu skall resursägaren också få autentisera sig. Detta sker när klienten fått svar och godkännande från servern. Då skickas resursägaren till

serversidans *owner authorization endpoint*. Här följer klientens nya temporära identifiering med och eventuellt en *callback* parameter.

4. På *owner authorization endpoint* låter server resursägaren autentisera sig. Exempelvis genom att låta den logga in med användarnamn och lösenord. Därefter genererar servern en verifieringsnyckel, en nyckel som skall vara omöjlig att gissa. Denna nyckel och klientidentifieraren, som skickades med användaren, läggs till parametrar i svaret till klienten. Resursägaren blir omdirigerad tillbaka till klienten med svaret.

5. Nu har både klienten och resursägaren autentiserats sig. Nu vill klienten få tag på ett par *token credentials*. Förfrågan skickas till serverns *token request endpoint*. Med den skickas verifieringsnyckeln som togs emot i förra steget och den temporära nyckeln som togs emot i steg 2. Signatur skickas också med, baserad på klienthemlighet och den temporära hemligheten. Svaret i nästa steg skickas i klartext, servern skall alltså begära att meddelanden i steg 5 och 6 skickas med säkra medel, exempelvis SSL/TLS.

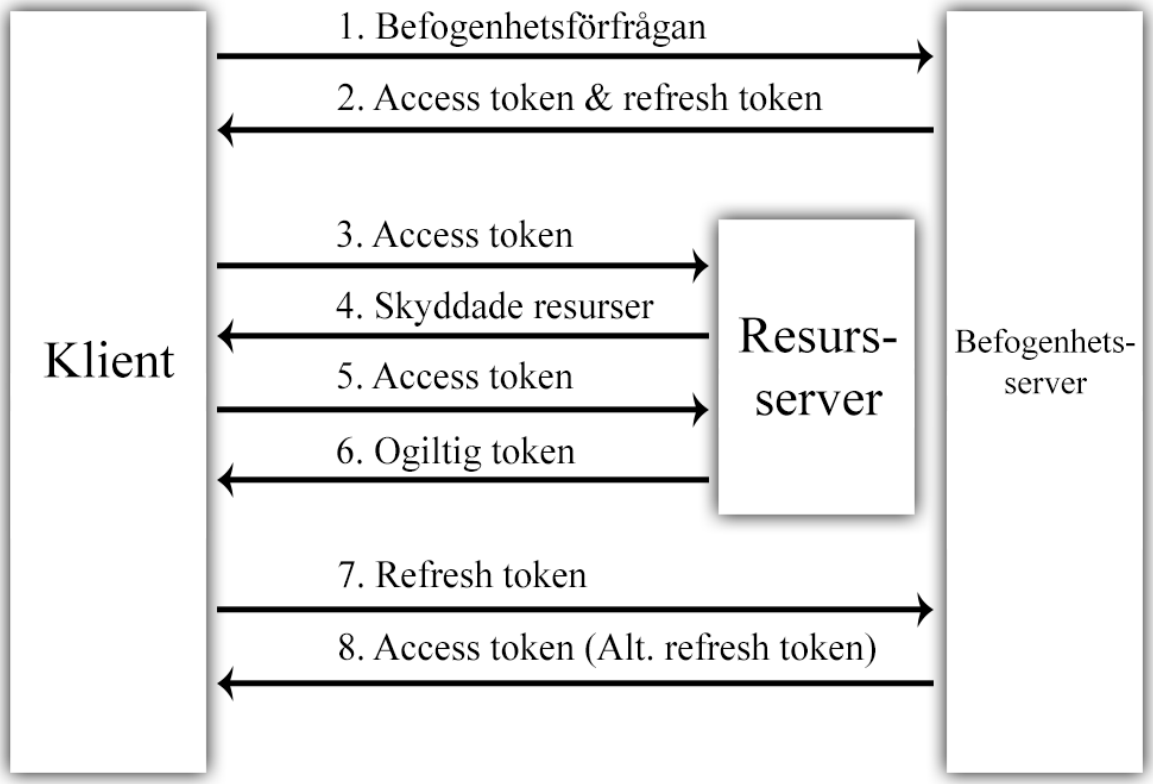
6. Servern kontrollerar att alla parametrar finns med, att resursägaren har autentiserat sig, om den temporära nyckelns livstid har passerats och om den använts innan samt om verifieringsnyckeln stämmer. Om allt är som det skall svarar servern med två nya nycklar, *token credentials*, identifierar och hemlighet. Servern sparar undan information om nycklar, klient, tidsstämplar mm.

7. De nya nycklarna kan nu användas av klienten för att skicka förfrågningar om resurser i resursägarens namn. Nycklarna används för att beräkna signaturen och autentisera sig som en pålitlig klient. Hur denna kommunikation upprättas och utförs specificeras inte av OAuth 1.0a-specifikationen, utan är upp till servern att bestämma.

2.4 Om OAuth 2.0

Detta delkapitel går igenom hur resursägaren, klienten och servern kan bära sig åt för att kommunicera med varandra i ramverket OAuth 2.0.

2.4.1 OAuth 2.0 överblick



Figur 3: OAuth 2.0 flödesschema

OAuth 2.0 har samma mål som OAuth 1.0a, att ge en tredje part tillgång till resursägarens skyddade resurser. Men den nya versionen har helt andra flöden och använder inte signaturer och förlitar sig helt på säkerheten i SSL/TLS. Figuren ovan ger en överblick över hur det fungerar.

Servern från figur 1 är i figur 3 uppdelad i en resursserver och en befogenhetsserver. Denna uppdelning är dock inte ett tvång, utan i många fall är dessa två samma.

1. Först autentiserar klienten och resursägaren sig mot befogenhetsservern. Autentiseringen kan ske på ett antal olika sätt, i kapitel 2.4.3 kommer dessa gås igenom.

2. Om autentiseringen går bra blir klienten tilldelad en *access token* och eventuellt en *refresh token*. Klienten har nu möjligheten att komma åt de skyddade resurser som resursägaren har godkänt.

3. Klienten kommer åt de skyddade resurserna genom förfrågningar till servern. Med dessa skickas den *access token* som tilldelades tidigare.

4. Om allt står rätt till kommer servern svara med de skyddade resurserna som klienten har frågat efter.

5. Precis som i steg 3 ber klienten om resurser med hjälp av sin *access token*.

6. Ett felmeddelande skickas tillbaks till klienten. Till exempel om *access token* inte stämmer eller om dess livstid har passerats fås felmeddelandet “invalid access token”.

7. Om *refresh tokens* stöds av servern och autentiseringsflödet så kan en förfrågan skickas om att tilldelas en ny *access token*. Med förfrågan skickas sin *refresh token*.

8. Om servern anser att allt stämmer skickas en ny *access token* och en ny *refresh token*.

2.4.2 Förberedelser

För att en klient skall kunna erbjuda en resursägare några OAuth-tjänster, måste den registrera sig hos en server. När en sådan registrering sker blir klienten tilldelad *client credentials*: klientidentifierare och en delad hemlighet. Klienten använder dessa för att autentisera sig mot befogenhetsserver. Eventuellt kan servern be klienten definiera en URI, var servern skall skicka sina svar. Om detta inte specificeras måste denna skickas med i klientens förfrågningar.

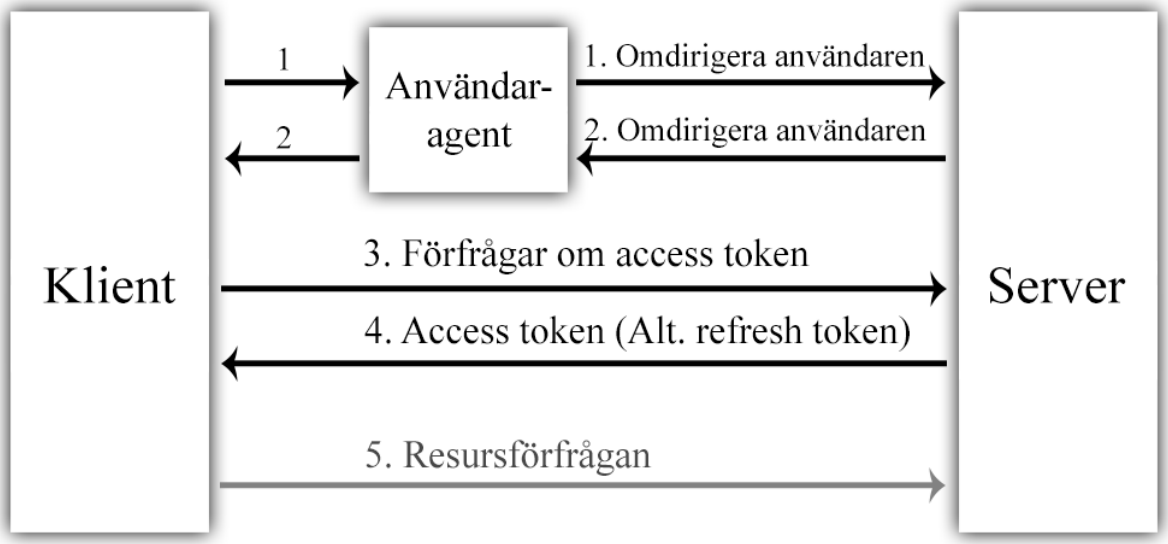
2.4.3 Autentiseringsflöden

Här beskrivs de fyra olika flödena för att dela ut befogenhet, som tas upp i specifikationen för OAuth 2.0[2]. Dessa flöden är alltså en förklaring av steg 1 - 2 i figur 3.

LinkedIns implementation av OAuth 2.0 använder endast autentiseringsflödet *authorization code grant*. Eftersom denna rapporten riktar in sig mot LinkedIn kommer störst vikt läggas på det flödet.

2.4.3.1 Authorization Code Grant

Detta flöde stödjer åtkomsten av både *access tokens* och *refresh tokens*. Flödet är även optimerat för klientens konfidentialitet, då ingen känslig information delas med användaragenten. Det är även baserat på vidarebefordring av resursägaren, vilket kräver att klienten skall ha möjligheten att kommunicera med resursägarens användaragent.



Figur 4: Authorization code grant-flöde

1. I början av autentiseringsflödet vidarebefordrar klienten användaragenten till en autentiseringsändpunkt (*authorization endpoint*) för autentisering. Ändpunkten är oftast en hemsida som ligger på befogenhetsservern. I detta skede skickar klienten med sitt klientid samt inkluderar vilken svarstyp som skall användas, i detta fall *code*, i URI:n. Alternativt kan klienten även skicka med ett antal andra parametrar i sin URI, *scope*, *state* och *redirect*. *Redirect* parametern definierar vart användaragenten skall vidarebefordras efter autentiseringen av resursägaren, detta kan vara förbestämt vid registrering.

2. I detta steg autentiserar resursägaren sig mot befogenhetsservern, vanligen genom att logga in med sitt användarnamn och lösenord. Användaragenten kontrollerar även att resursägaren tillåtit klientens förfrågan om access till de skyddade resurserna, oftast genom att resursägaren får trycka på en godkännknapp på hemsidan.

   Om resursägaren tillåter klienten access, så vidarebefordras användaragenten tillbaka till klienten. Detta sker antingen med en förbestämd URI parameter eller genom användning av den som

skickades med från klienten i steg 1. I URI:n inkluderas behörighetskoden, som befogenhetsservern genererat, och eventuellt den state parameter som klienten skickade med i steg 1. Klienten byter i senare steg behörighetskoden mot en access token och en eventuell refresh token.

Om state parametern som klienten skickade med i steg 1 är densamma som den får tillbaka från befogenhetsservern, kan klienten vara säker på att ingen CSRF-attack har skett, om inte avbryts flödet.

3. Klienten ställer en förfrågan till befogenhetsservern om att få en *access token* och eventuell *refresh token*. Klienten autentiserar sig mot befogenhetsservern, med hjälp av sina *client credentials*. För att verifiera förfrågan inkluderar klienten befogenhetskoden som den fick i steg 2 och vilken typ flöde man använder sig av. Om parametern *redirect* använts i steg 1 skall den också inkluderas.

4. Befogenhetsservern autentiserar klienten, validerar behörighetskoden och eventuellt kontrollerar att redirect-parametern är densamma som användes för att vidarebefordra klienten i steg 2. Om allt stämmer överens så tilldelar befogenhetsservern klienten en *access token* och eventuell *refresh token*.

5. Klienten kan nu ställa förfrågningar om de skyddade resurser som finns på resursservern i resursägarens namn, med hjälp av den *access token* som blivit tilldelad.
   Om klienten blivit tilldelad en *refresh token* kan klienten få tillgång nya *tokens* utan att behöva göra om flödet.

2.4.3.2 Implicit grant

Det implicita flödet stödjer endast åtkomsten av access tokens och är optimerat för publika klienter, exempelvis klienter inbyggda i webbläsaren. Detta flöde är även “som det tidigare” baserat på vidarebefordring av resursägaren, vilket kräver att klienten skall ha möjligheten att kommunicera med resursägarens användaragent.

En av de större skillnaderna mellan detta flöde och *authorization code grant* är att access tokens är synliga för användaragenten ett litet tag, då de skickas med i URI:n när resursägarens autentiserat sig. Det sker ingen autentisering av klienten utan hela flödet bygger på svarsadresser som är registrerade i förväg.

2.4.3.3 Resource owner password credential grant

Detta flöde skall endast användas där resuresägaren kan lita på klienten, och att den säkert kan hantera sina *credentials*. Resursägarens *credentials* kan till exempel vara ett användarnamn med tillhörande lösenord. Själva flödet går ifrån grundidén med OAuth, där resursägaren inte skall behöva dela med sina *credentials* till en tredjepart.

Flödet för att få befogenhet är inte lika komplicerat som de två tidigare. Resursägaren får logga in direkt på klienten istället för att bli vidarebefordrad till servern. Klienten skickar själv resursägarens *credentials* till befogenhetsservern, som sedan skickar tillbaks en *access token*.

2.4.3.4 Client credentials grant

Detta flöde används då klienten och resursägaren är densamma. Flödet baseras på att klienten autentiserar sig mot befogenhetsservern med sina *client credentials*. I specifikationen för OAuth 2.0[2] står det att utvecklare kan specificera andra metoder för autentiseringen av klienten. Flödet skall endast användas av konfidentiella klienter, exempelvis en klient implementerad på en säker server med begränsad access till klientens *credentials*.

2.5 Joomla

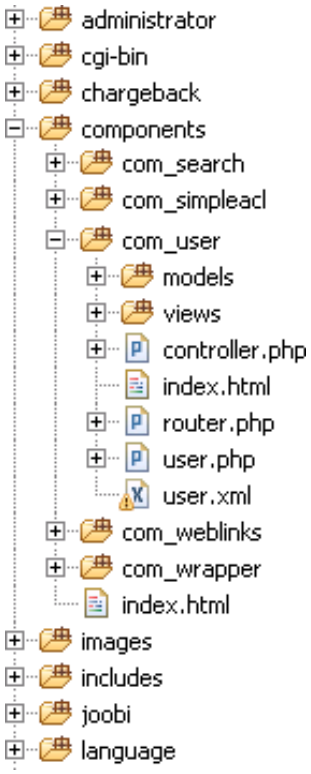
PMCGs hemsida är uppbyggd i Joomla, som är ett så kallat *content management system* (CMS). Ett CMS tillåter smidig central hantering av innehållet på en hemsida eller liknande applikation. Dessutom behöver den som skapar hemsidan inte vara en så kallad avancerad användare, alltså någon som faktiskt kan utveckla exempelvis hemsidor. I ett CMS finns ofta en tydlig struktur för vad filer och mappar skall heta och var de skall befinna sig.

Joomla är byggd på programmeringsspråken PHP och JavaScript samt använder sig av MySQL databaser.

Fördelen med att bygga en sida i Joomla är att den blir väldigt modulär. Varje del på sidan skapas separat och jobbar självständigt. Detta gör det enkelt att lista alla delar för administratören och att aktivera eller ändra specifika moduler utan att påverka resten av hemsidan. Detta gör det också enkelt att ändra designen på hemsidan utan att störa funktionaliteten.

En annan fördel med att skapa moduler är att de kan exporteras till andra Joomla-sidor och användas där utan större problem. Det finns många tusentals olika nedladdningsbara moduler och designteman gratis på Joomlas hemsida och fler skapas hela tiden.

Joomla har dock fått mycket kritik under åren. Strukturen är väldigt komplex och tar mycket tid att lära sig att hantera. Att skapa till exempel en *component* kräver att ett stort antal mappar och filer skapas med rätt namn och i rätt trädstruktur.



Figur 5: Exempel på Joomla-mappstruktur

I figur 5 visas ett exempel på en del av mappstrukturen i Joomla. Komponenter skall heta com_NAMN. Under denna skall minst filerna index.html, controller.php och NAMN.php finnas. Det skall också existera en mapp vid namn views och under den kommer det ytterligare filer och mappar.

2.6 PHP: Hypertext Preprocessor

Programmeringsspråket PHP används huvudsakligen inom webbprogrammering på serversidan. Utöver en del egna PHP-specifika funktioner lånas mycket av syntaxen från C, Java och PERL, vilket gör att språket är lätt att förstå och lära sig för den som reda kan programmera. PHP är ett väl dokumenterat språk med väldigt tydliga funktionsbeskrivningar och typbeskrivningar samt kodexempel.

PHP kan användas för att göra dynamiska sidor. Istället för att en sidas utseende och innehåll är förbestämt kan sidan byggas upp när den skall visas för någon. Vilken text som skall visas hämtas från en databas och en del knappar visas endast om användaren har loggat in på hemsidan. Först när

sidan har konstruerats av PHP skickas HTML-koden till användarens användaragent.

2.6.1 PHP-exempel

I figur 6 visas ett enkelt exempel på hur en sida på en hemsida kan skapas med hjälp av PHP:

```
<html>
<?php
// function that returns the body text
function getBodyText(){
    return "<h1>Hello PHP world!</h1>";
}

// load the head and body
$head = "<head> <title>Example</title> </head>";
$body = "<body>" . getBodyText() . "</body>";

// print the head and body
echo $head;
echo $body;
?>
</html>
```

Figur 6: Exempel på PHP-kod

PHP-taggen, *<?php .. ?>*, omsluter all kod som skall tolkas som PHP-kod. Först sätts en variabel *$head* till en statisk text. Därefter sätts variabeln *$body* till vad funktionen *getBodyText()* ger ifrån sig, vilket är en HTML-headline med texten “Hello PHP world!”. Sist skrivs *$head* och *$body* ut som HTML-kod. Resultatet blir en HTML-sida med titeln “Example” och en stor text “Hello PHP world!”.

3 Metod

3.1 Teoretisk Analys

3.1.1 insamling av information

För att få förståelse för de båda versionerna, har specifikationerna[1,2] för OAuth 1.0a och 2.0 lästs och analyserats. Utöver detta har aktuell litteratur[4] om ramverket granskas. Mycket information har även inhämtas från OAuths egna hemsida[15], samt på diverse bloggar och forum.

3.1.2 Analys av insamlad information

För att kunna jämföra de två versionerna valdes ett antal punkter ut: praktisk implementation, dokumentation och säkerhet. Valet av den första punkten är självklar, eftersom mesta av kritiken riktad mot den första versionen var protokollets svårigheter vid implementering. Målet var att få reda på hur svårt det egentligen är och om den nya versionen löser problemet. Den andra punkten valdes då dokumentation är en mycket viktig del inom utveckling. Om det finns dålig dokumentation blir det praktiska arbetet mycket svårare och risken för slarv ökar. Säkerhetspunkten var också ett självklart val, då det är ett av rapportens huvudsyften att undersöka just denna punkt. Det var också viktigt att få en bild av säkerhetsbristerna och hur väl dokumenterade de är.

Under varje punkt ställdes för- och nackdelar från den insamlade datan mot varandra. En muntlig diskussion fördes också för att ytterligare brister och fördelar skulle uppdagas.

3.1.3 Analys av LinkedIns lösningar

Ett av målen i denna rapport var att skapa en implementation mellan PMCGs projektportal och LinkedIn. Efter säkerhetsanalysen gjordes därför en separat analys av LinkedIns lösningar av de båda versionerna. Här användes kunskapen från den tidigare analysen för att göra en bedömning av LinkedIns implementationer.

Tillgång till LinkedIns serverimplementation saknas, men en god bild om LinkedIns OAuthimplementationer kunde ändå bildas. Utifrån de parametrar som klienten måste använda samt den dokumentation[14] som finns på LinkedIns hemsida kan man kontrollera vilka val i säkerhetsfrågorna som gjorts. Detta ansågs tillräckligt för att en analys skulle kunna utföras.

3.1.4 Presentation och möten för/med företag

En presentation av protokollet och ramverket samt de analyser som gjorts hölls för PMCG och examinator. Målet med denna presentation var att diskutera vilket av de två versionerna som var bäst lämpade för PMCG. Ett antal möten fördes också med PMCG för att få en bild av hur de såg på säkerhet kontra användbarhet.

3.1.5 Implementation

3.1.5.1 Versionshantering

För versionshantering har Subversion (SVN) använts. Först skickas ett projekt, i detta fall PMCGs hemsida, upp på en SVN-server, sedan kan denna version laddas ned till en lokal maskin och ändringar kan göras. Dessa skickas sedan tillbaka och appliceras på koden på servern, men med nytt versionsnummer. Detta gör det möjligt för utvecklare att återgå till en tidigare version. Ändringar på samma version från olika maskiner kan automatiskt slås ihop.

3.1.5.2 Utvecklingsmiljö

Själva implementationen skedde genom användningen av mjukvaran WMware. WMware tillåter access till virtuella maskiner från vilken dator som helst. Utvecklare kan därför använda dessa för utveckling utan att vara låsta vid en och samma plats.

I den virtuella miljön har flera program använts för programmering och testkörning av hemsidan. Själva hemsidan har hämtats från Subversion. För att köra hemsidan på lokal maskin användes XAMPP. Med XAMPP skapas en lokal webbserver med stöd för bland annat PHP och MySQL-databaser, vilket krävs för att köra en Joomlahemsida.

Programmet Eclipse har använts för att skriva all kod. Det är en gratis utvecklingsmiljö med öppen källkod som framförallt är inriktad för javaprogrammering. Tilläggsprogram kan laddas ner så den även stödjer bland annat PHP. Eclipse erbjuder många fördelaktiga funktioner under programmering. Radnumrering, färger i koden och överblick över funktioner i klasser är några exempel. Snabb och enkel sökning till funktioners källkod/deklarering, erbjuds även av Eclipse. Denna funktion är mycket användbar för att snabbt förstå strukturen hos Joomla. Utvecklare slipper då spendera tid på att leta upp specifika filer/metoder.

3.1.5.3 utvecklingsmetod

Under implementationen användes parprogrammering. Två utvecklare sitter vid samma arbetsstation och byter av varandra, så att en av utvecklarna skriver kod och den andra granskar koden samt kommer med synpunkter och kritik.

4 Analys

I denna del sammanställs den teoretiska kunskap som samlats in från litteratur och praktiskt arbete. Protokollet och ramverket behandlas först och därefter analyseras LinkedIns implementationer av dem.

Resursägarens upplevelse är densamma för de båda versionerna, så utvecklaren som skall implementera väljer den metod som behärskas eller den mest lämpad för ändamålet samt upprätthåller eventuella säkerhetskrav.

4.1 Praktisk implementation

Här analyseras svårigheter som man kan stöta på vid en implementation av OAuth 1.0a eller OAuth 2.0.

4.1.1 OAuth 1.0a

Implementationer av OAuth 1.0a har enligt många utvecklare setts som ett krävande arbete, då användandet av signaturer krävs. I många fall besitter inte klientsidans utvecklare dessa kunskaper, då hen kan vara allt från säkerhetsexpert till "hobbyprogrammerare". Detta var en av de stora anledningarna till att en ny version av OAuth utvecklats, vilket ledde fram till version OAuth 2.0.

OAuth 1.0a är ett protokoll, vilket betyder att det finns väldigt tydliga regler på vad som får göras. Detta leder till att alla serverar som har implementerat OAuth 1.0a har liknande lösningar, så när en klient har lyckats koppla sig till en specifik server, är det lätt att koppla sig till ännu en. Risken finns dock att protokollets regler inte riktigt passar den lösning man tänkt sig, vilket då leder till extra arbete.

Token credentials har mycket lång livstid i OAuth 1.0, i vissa fall har de en oändlig livstid. Detta gör hanteringen av nycklar mycket lättare, eftersom inga nya nycklar behöver hämtas. Detta sparar både plats och eventuellt exekveringstid i databaser.

4.1.2 OAuth 2.0

En klientsida i OAuth 2.0 anses vara lättare att implementera än OAuth 1.0a, då utvecklare slipper signaturer och flödet kan kännas enklare och mer logiskt. Det arbete som istället läggs till är hantering av certifikat, som följd om all säkerhet inom SSL/TLS skall utnyttjas.

OAuth 2.0 kunde inte klassas som ett protokoll, då det finns många valmöjligheter till olika server- och klientimplementationer. Det fick då istället klassas som ett ramverk. En följd av detta blir att olika server- och klientimplementationer kommer vara väldigt olika varandra. Detta betyder också att återanvändbarheten av en lösning minskar kraftigt. Att skapa en klientsida till ett API har dock ett mer logiskt flöde och kräver mindre kunskaper, och anses då enklare att implementera, jämfört med OAuth 1.0a.

OAuth 2.0 är inte bakåtkompatibel till OAuth 1.0a, så ett nytt API måste skapas från grunden om man vill uppgradera sin applikation från den tidigare versionen.

I teorin har *client credentials* mycket kort livstid, vilket leder till extra arbete i hantering av sina nycklar, eftersom gamla nycklar måste uppdateras.

Att skapa ett OAuth 2.0 API kräver att utvecklarna förstår de olika flöden och väljer de som passar bäst till ändamålet och utifrån eventuella säkerhetskrav. Det finns många val under varje flöde som är frivilliga. Säkerhetsbrister kan förekomma och som följd kanske utnyttjas, om kunskap om flödena och de val som skall göras under dessa saknas. Om flera flöden implementeras i samma API finns risken att de som skall koppla upp sig, klienten, väljer fel flöde. Som nämnts i kapitel 4.1.1 är inte alltid klientsidan skapat av en expert, vilket ökar risken för misstag.

4.1.3 Slutsats

Den som skall implementera ett server-API i OAuth 2.0 står inför en hel del val, eftersom mycket är valfritt. Detta kan medföra mycket arbete för utvecklarna och om tillräcklig kunskap saknas kan det leda till många säkerhetsbrister. När valen väl är gjorda och det är dags att skriva kod, är OAuth 2.0 att föredra. Det mesta av säkerheten sköts av SSL/TLS. Det innebär att koden handlar mer om hur de skall hantera trafiken som tas emot än huruvida den stämmer eller inte, som med OAuth 1.0a. Där måste signaturer beräknas och tidstämplar och *nounce* hanteras.

Den som implementerar klientsidan har det lättare oberoende av OAuth version, eftersom alla val kring säkerhet redan är gjorda. Här handlar det om att ta reda på vilka parametrar som skall skickas med och hur de skapas. OAuth 2.0 skickar färre variabler som dessutom är lättare att skapa, då de flesta skall vara statiska. I den första versionen måste signaturer beräknas, och en del andra variabler som tidsstämplar och *nounce* måste genereras. Nackdelen med den senare versionen är att den kräver certifikatshantering, vilket anses vara besvärligt[5].

Värt att nämna är att det finns flera OAuth 1.0a bibliotek som gör det mesta åt användaren. Om ett sådant bibliotek används blir OAuth 1.0a inte alls lika svårt att implementera. Det har dock varnats för att många bibliotek som finns ute inte riktigt håller måttet[9], då de kan vara väldigt riktade för just ett API men marknadsförs som att det är skapat till alla.

4.2 Dokumentation

I denna del utreds vilken typ av dokumentation som finns kring de båda versionerna.

4.2.1 OAuth 1.0a

Specifikationen till OAuth 1.0a anses vara dåligt skriven, bland annat av dess egen författare[9]. Det är svårt att få en övergripande blick av hur protokollet fungerar. Mycket tekniska exempel som hjälper den som redan har förstått hur flödet går till. Bilder som visar överblickar saknas också.

Författaren, Eran Hammer, av specifikationen har en blogg, http://hueniverse.com, där han har samlat mycket information kring denna version. Han behandlar en mängd olika delar av OAuth 1.0a i sina artiklar bland annat historia, säkerhet och brister samt exempel.

De som har skapat en server-API har egen dokumentation av just sina egna implementationer. Är denna ordentligt skriven skall inte klientutvecklaren behöva läsa någon annan dokumentation.

Protokollet är gammalt och därför har utvecklare haft gott om tid att ställa frågor på internet och fått svar. Kvalitén på dessa svar varierar dock, som kan förväntas.

4.2.2 OAuth 2.0

Specifikationen till denna version bedöms vara bättre skriven än sin föregångare. Utöver ett mer logiskt upplägg av specifikationen finns också bilder som ger övergripande bild av ramverket samt bilder som går ner på djupet.

Utöver den digitala dokumentationen på internet har också en del böcker skrivits om OAuth 2.0, exempelvis *Getting Started with OAuth 2.0* av Ryan Boyd år 2012 och *OAuth 2.0: The Definitive Guide* av Aaron Parecki år 2013.

Ramverket är mycket nyare än sin föregångare därför finns det heller inte alls lika mycket om det på internetforum. Eftersom det är OAuth 2.0 som

marknadsförs nu, istället för ettan, kommer dokumentation i form av forumstrådar att växa fort inom den närmsta tiden.

4.2.3 Slutsats

Dokumentationen till den första versionen är ganska spridd. Utöver den dåliga specifikationen för OAuth 1.0a hittas mycket av informationen på olika forum och bloggar. OAuth 2.0 har en mycket bättre specifikation och ett antal böcker har släppts eller kommer att släppas. Förutom detta växer antal forum- och blogginlägg inom detta ämne eftersom det är den versionen som marknadsförs av de stora företagen just nu, exempelvis Google och Facebook.

4.3 Säkerhet

Är man intresserad av alla säkerhetsbrister i de båda versionerna finns det mycket information utlagt. Till OAuth 2.0 finns det ett stort dokument tillägnat just potentiella attacker mot ett sådant system[3]. Här analyseras säkerhetsbristerna hos OAuth 1.0a för att sedan se om OAuth 2.0 har löst dessa, och om det uppkommit nya hot.

4.3.1 OAuth 1.0a

4.3.1.1 Signaturer i OAuth 1.0a

Tack vare signaturerna räknas OAuth 1.0a som ett mycket säkert protokoll. Endast den som har tillgång till de hemliga *credentials* kan skapa en signatur. Signaturen byggs också upp av förfrågans parametrar, ett slumpmässigt genererat *nounce* och en tidsstämpel. Detta leder till att varje signatur blir unik. Så om någon lyckas fånga upp ett meddelande kan inte signaturen återanvändas, så länge originalmeddelandet kommer fram till servern först. Har också servern och klienten på förväg bestämt vart alla svar skall skickas har man skyddat sig helt från så kallade replay-attacker.

I specifikationen för OAuth 1.0a specificeras användningen antingen av algoritmen *RSA-SHA1* eller *HMAC-SHA1* för signering av meddelanden i autentiseringsflödet. Professor Wang Xiaoyun har funnit en metod att hitta kollisioner i hashalgoritmen *SHA-1*[11] relativt fort, detta leder till att säkerheten hos algoritmer som använder sig av *SHA-1* minskar. Detta har dock inte direkt påverkan på säkerheten hos *HMAC*, eftersom den bygger sin säkerhet på mer än hashens kollisioner. Detta anses dock som en varning att det kanske är dags att börja fundera på en annan metod än *SHA-1* i allmänhet, och då även inom OAuth-världen. *National Institute of Standards and Technology(NIST)*[11] skrev att användningen av *SHA-1* skall ha upphört till år 2010.

Om *RSA-SHA1* används ökar säkerhetskraven på klienten. När klienten skapar en signatur med hjälp av *RSA* används endast en nyckel, som tilldelas i sista steget i autentiseringsflödet. Om en attack leder till att någon får ut denna nyckel så är allt förlorat. I *HMAC-SHA1* används istället två nycklar som klienten har fått vid två olika tillfällen, först när den registrerar sig, klienthemlighet, och sedan när den får sina *token credentials*.

4.3.1.2 Odefinierat i OAuth 1.0a

Det finns delar av OAuth 1.0a som inte är fullständigt definierade. Dessa kan ställa till med problem beroende på hur utvecklarna väljer att lösa dem.

Specifikationen har inte specificerat hur de skyddade resurserna skall skickas mellan server och klient. Detta kan leda till att en del implementationer skickar data öppet, vilket inte ger något skydd mot enkla attacker som *eavesdropping*. Skall data skickas öppet kan man också ifrågasätta varför ens OAuth används. För att skicka sina resurser hemligt kan olika krypteringsmetoder användas eller SSL/TLS.

Det finns inget i OAuth 1.0a som specificerar spannet av resurser, alltså vilka skyddade resurser som resursägaren ger klienten tillgång till. Om detta glöms bort eller ignoreras av serverns utvecklare leder det till att klienten har tillgång till alla ägarens resurser.

Det finns inget krav på livslängd på *client credentials*, vilket har lett till att vissa implementationer delar ut tokens som lever för alltid. Ett exempel på detta är Twitter. Faran i detta är att den som har fått tag på *credentials* kan för alltid komma åt ägarens skyddade resurser, tills ägaren säger stopp. Alternativt kan klienten få möjligheten att säga upp sina egna tokens.

4.3.2 OAuth 2.0

Det utvecklarna till OAuth 1.0 hade i åtanke när den skapades var att på ett säkert sätt berätta om du har befogenhet till någons resurser över en osäker kanal. Detta ledde till den krångliga lösning som finns idag. De som istället förespråkar OAuth 2.0 menar att denna komplexitet inte är nödvändig idag så länge man använder sig av SSL/TLS, då detta ger en säker förbindelse mellan två punkter.

4.3.2.1 Inga signaturer

SSL/TLS är mycket säkert, om man använder sig av de senaste versionerna. Så om klienten och servern kan använda en SSL/TLS-förbindelse kommer ingen åt trafiken som skickas mellan dem. Detta gör det möjligt för OAuth 2.0

att skicka sina hemligheter mellan klient och server, till skillnad från ettan där de hemliga nycklarna aldrig skickas. Problemet ligger dock i och med att OAuth 2.0, precis som OAuth 1.0a, inte kan känna igen vem den talar med. Det är då mycket möjligt att utföra en MITM attack. I Oauth 1.0a blir inte konsekvenserna lika grova, då inga hemligheter skickas, men nu skickas klientens access token. Denna token är allt som krävs för att komma åt resursägarens resurser, om inte servern har implementerat ytterligare säkerhet för att avgöra vem den talar med.

4.3.2.2 Riskerna i OAuth 1.0a

Huruvida de risker som tidigare togs upp i denna rapport om OAuth 1.0a finns kvar utreds i denna del.

I OAuth 1.0a specificerades aldrig hur de skyddade resurserna skulle skickas, vilket kunde leda till osäkra lösningar. I OAuth 2.0 krävs dock anslutning med SSL/TLS, vilket skyddar all data som skickas mellan ändpunkterna.

Scope, en parameter som lagts till i OAuth 2.0, kan skickas med under autentiseringsflödet för att specificera vilka resurser som klienten vill komma åt. Dessa resurser borde listas för resursägaren innan den godkänner klienten. Även om denna parameter är alternativ finns den med i specifikationen, vilket kommer leda till att olika servrar kommer använda samma lösning.

I denna version har möjligheten med livslängder på *access tokens* ökat i och med *refresh tokens*. I *OAuth 2.0 Threat Model and Security Considerations*[3] rekommenderas en kort livstid på sina tokens, detta för att minska skadorna om en access token skulle läckas. Dock om en erfaren inkräktare får tag på en *access token* krävs inte speciellt lång tid för att orsaka tillräckliga skador, till exempel skicka en resursförfrågan om känslig eller all information.

4.3.2.3 nya risker

Då OAuth 2.0 är relativt nytt, så det kan finnas säkerhetsbrister i ramverket som ännu inte har upptäckts. En del implementationer har visat sig ha säkerhetsbrister och har blivit åtgärdade[12], men åtgärdas endast i implementationen och inte i specifikationen. Eftersom varje OAuth 2.0 lösning är så unik är det svårt att lägga till säkerhetsåtgärder i specifikationen, vilket leder till att nya servrar kan komma att begå gamla misstag.

När en part implementerar ett API med användning av OAuth 2.0, finns risken att de endast tar med de absolut mest nödvändiga parametrarna till autentiseringsflödet. Problemet som ligger i specifikationen för OAuth 2.0 är att många säkerhetsparametrar är klassade som antagligen *recommended* eller

optional istället för *required,* vilket kan leda till en osäker implementation av ett OAuth 2.0 API.

I OAuth 2.0 kan servern tillåta möjligheten att använda sig av dynamiska svarstyper, när man skickar en förfrågan väljer man vilket flöde man vill följa. Ett annat möjligt val är att låta klienten välja var svaret skall tas emot vid förfrågan. Om båda dessa val är dynamiska i samma API har det visat sig svaga mot så kallade cross site scripting (XSS) attacker[12].

En stor risk som har pekats ut i denna version är hanteringen av certifikat som krävs. För att SLL/TLS skall vara så säkert som möjligt måste ens program klara av att hantera en stor mängd felmeddelanden. SSL/TLS fungerar så att när något går fel i någon certifiering så avbryts inte kopplingen, utan alla parter varnas endast om att någonting har gått fel. Detta gör det möjligt för serversidan/klientsidan i OAuth 2.0 att fånga upp alla felmeddelanden och strunta i dem. Det finns flera exempel på detta eftersom det är så pass mycket lättare än att lösa problemet med certifikaten. Denna risk innebär att en bra implementation av serversidan inte räcker utan denna sida får hoppas att alla klienter faktiskt gör som de ska. Serverns säkerhetsansvar har flyttats över på klienten.

4.4 LinkedIn

Här analyseras LinkedIns implementation av OAuth 1.0a och 2.0.

LinkedIn förbjuder användandet av iframes för både OAuth 1.0a och OAuth 2.0. Vill en programmerare bädda in ett HTML-dokument i ett annat används en iframe, till exempel vill man kanske visa reklam från en annan hemsida. Att sätta stopp för iframes ger extra skydd mot så kallade *clickjacking* attacker, då skadlig och dold kod kan läggas in i en iframe och utföra oönskade handlingar.

LinkedIn har en variabel som de kallar för *scope* som är oberoende av OAuth version. Den kan skickas med från klienten när användaren skall autentisera sig och godkänna klienten. Om denna variabel inte skickas med sätts befogenheterna till *basic profile,* som ger befogenheten att komma åt namn, användarid på LinkedIn, profilbild samt lite annan information om personen i fråga.

Token eller access token lever i 60 dagar på LinkedIn. Så både klienten eller någon illasinnad som får tag på dessa nycklar har gott om tid att få ut all information de har befogenhet att få ut.

4.4.1 LinkedIn Oauth 1.0a

OAuth 1.0a är ett protokoll och har därmed strikta regler som LinkedIn måste följa. Det innebär att valmöjligheterna inte var så många när de implementerat sitt API för ettan. Det finns dock som tidigare nämnt odefinierade delar i protokollet, och dessa undersöks i rapporten.

LinkedIn skriver att de räknar med att de flesta implementerar den nya versionen eftersom den är mycket enklare än sin föregångare, men att de tänker behålla sin dokumentation för de som redan har en lösning. Denna dokumentation är lite svår att följa om man jämför med den nya versionen. Den har flera sidor som inte verkar följa en viss ordning, men alla behövs för att man skall kunna göra sin implementation. Det finns länkar som inte fungerar och kodexempel som man inte riktigt vet om de behöves eller ej. Dock kan detta komma att åtgärdas i framtiden, men under skrivandets stund är detta fallet.

Klienten kan med sina förfrågningar skicka med vart servern skall svara till. I ett sådant fall kommer den fördefinierade URI inte att användas. Om en inkräktare har möjligheten att konstruera godkända förfrågningar kan denna se till att svaren skickas till sig.

4.4.2 LinkedIn Oauth 2.0

LinkedIns dokumentation om sin OAuth 2.0 lösning är kortfattad och tydlig. De få sidor som finns sammanfattar all den information som kan tänkas behövas och presenteras i ett logiskt flöde. Dessutom innehåller den fullständiga kodexempel.

LinkedIns implementation använder sig endast av autentiseringsflödet *authorization code grant*. Detta göra att de endast kommer ta emot förfrågningar med svarstypenen *code*. Att LinkedIn endast tar emot en svarstyp minskar risken för att cross-site scripting (XSS) attacker skall lyckas.

LinkedIn använder *access tokens* som *bearer tokens*. Att använda *bearer tokens* är en risk, då de inte kräver någons sorts autentisering av den som ställer förfrågan. Om någon får tag på en *bearer token* kan hen få ut all information om den riktiga ägaren. Genom att länka ihop *access token* med en digitalsignatur hade den potentiella risken för att olovlig användning minskats markant.

LinkedIn stödjer inte användandet av *refresh token*, så om en *access tokens* livstid passerats måste hela autentiseringsflödet göras om. LinkedIn behöver då inte bekymra sig om att någon får tag på en *refresh token* och kommer ha

tillgång till resurserna ända tills den stängs av. LinkedIn skriver att man dock kan ha specialkod på klientsidan som kan förnya ens *access token*, ifall livstiden håller på att passeras. I det fallet kommer inte användaren märka något, eftersom de redan gett sitt godkännande.

När klienter vill byta befogenhetskoden mot en *access token* kontrollerar LinkedIn att *redirect*-parametern, som användes för att få koden, är densamma som LinkedIn tagit emot. Risken för att LinkedIn skickar en *access token* till någon illasinnad klient eller användare minskar kraftigt på grund av detta.

LinkedIn kräver användandet av en *state*-parameter, vilket gör att LinkedIn blir mindre känsliga mot CSRF attacker.

Svarsadresser som är registrerade i förväg är inte något LinkedIn använder sig av i sin OAuth 2.0 lösning, vilket gör autentiseringsflödet mindre säkert. Någon som attackerar kan försöka ändra denna adress under flödet och få tokens för sig själv.

4.4.3 Slutsats

Det finns stöd för hur man skall kunna uppgradera till OAuth 2.0 om man så skulle vilja, men det är inte något som LinkedIn anser vara nödvändigt.

LinkedIn har gjort ett bra jobb i båda versionerna. Den användare som skall logga in får i och med *scope*-parametern se direkt vad den godkänner. Att de har en livslängd på sina tokens är också bra, men den kanske är lite för lång.

De har lyckats ganska bra med sin OAuth 2.0 lösning, det finns skydd mot mycket. Den stora nackdelen är att de inte identifierar klienten, utan en *access token* är möjlig att använda varifrån man än vill. Det skulle kunna vara möjligt att logga in på andra sidor med samma access token, trots att inte användaren har godkänt den sidan.

5 Resultat

5.1 Val av protokoll/ramverk

För implementationen hos företaget PMCG Scandinavia AB valdes OAuth 2.0. Nedan diskuteras valet av OAuth 2.0.

LinkedIns implementation klarar sig mot de flesta attacker som kan tänkas mot ramverket, genom att använda sig av parametrarna som togs upp i kapitel 4.4. Dessa val gör att LinkedIns implementation av OAuth 2.0 nästan har samma säkerhetsnivå som deras OAuth 1.0a. Vilket bidrar till att mer vikt kan läggas på andra punkter.

Analysen, kapitel 4.1, visar att implementeringen av OAuth 2.0 är mindre omfattande och mer hanterlig. Detta gör det också lättare för PMCGs framtida anställda att förstå och underhålla sin kod.

LinkedIns dokumentation för OAuth 2.0 var betydligt mer genomgående och fullständig än vad den var för OAuth 1.0a. Den är lätt att följa med och sprider sig inte över så många sidor, vilket minskar risken för missuppfattningar och fel. Detta leder också till att implementationstiden blir kortare och underhåll enklare.

OAuth 2.0 är det som marknadsförs just nu, och det är ofta en bra ide att följa med inom data- och IT-världen. Ramverket och dess användning kommer eventuellt att växa samt att ändringar i dess specifikation sker, för att öka säkerheten hos ramverket.

5.2 Implementation

5.2.1 Funktionalitet

Genom en testimplementation har inloggning via hemsidan LinkedIn möjliggjorts hos PMCG. En knapp har lagts till, för att kunna logga in via LinkedIn. Knappen vidarebefordrar användaren till LinkedIn för autentisering och godkännande, därefter vidarebefordras användare tillbaks till PMCGs hemsida. Om användaren redan är inloggad på LinkedIn, så vidarebefordras användaren direkt till PMCG, utan att behöva återautentiseras hos LinkedIn.

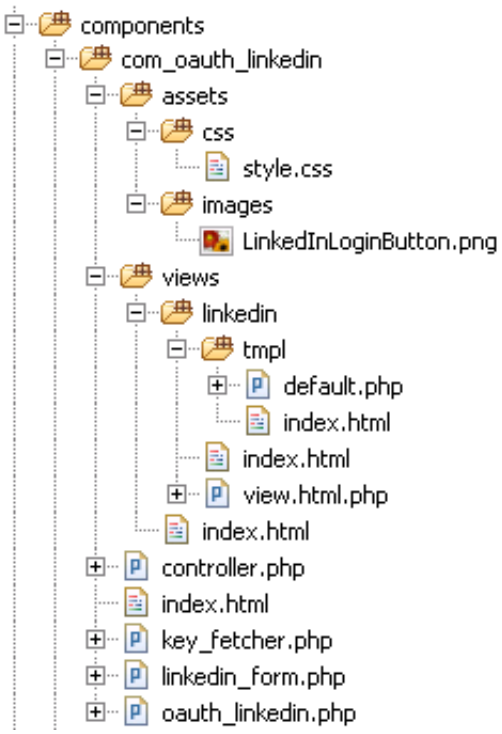
LinkedIns villkor för användning[13] dikterar att ingen användarinformation får kopieras utan användarens direkta godkännande. Med ett godkännande får klienter utföra lagring av aktuell användarinformation, denna information får dock inte förnyas utan användarens aktiva godkännande. I och med att det skall vara möjligt att registrera nya användare som brukar inloggningen via LinkedIn, så måste användarinformation lagras i PMCGs databas.

För att följa LinkedIns användningsvillkor vidarebefordras användare till en villkorssida hos PMCG, efter autentiseringen hos LinkedIn. Denna vidarebefordring sker endast första gången en användare loggar in via LinkedIn. På denna sida måste ett godkännande ges att PMCG skall få rättigheten att lagra dess användarinformation. Om användaren godkänner PMCGs villkor registreras LinkedIn-kontot som en ny användare i Joomlas databas.

På villkorssidan får även användaren möjligheten att koppla sitt nya LinkedIn-konto med ett redan existerande PMCG-konto. Detta för att en gammal användare skall kunna behålla sina rättigheter och annan kontospecifik data men ändå kunna utnyttja den nya inloggningsfunktionen. Denna koppling kommer skriva över inloggningsuppgifterna för PMCG-kontot, med information från LinkedIn. Så inloggning via LinkedIn blir obligatoriskt. Kopplingen kan i nuläget endast göras från villkorssidan, så kopplar inte användaren sitt konto vid detta steg kan det inte göras i senare.

5.2.2 Joomla component

Då testimplementationen skall fungera ihop med PMCGs hemsida så skapades inloggningen via LinkedIn som en självständig komponent till Joomla strukturen. Detta skapar möjligheten att smidigt underhålla implementationen. Dessutom kommer koden att ligga på en och samma plats. Eftersom den skapas som en Joomla komponent kan den även sättas på och stängas av med några enkla knapptryck. Strukturen på komponenten beskrivs av figur 7.



Figur 7: Mappstruktur för implementationen

Implementationen har dock inte helt kunna hålla sig till den nya komponenten. Koden för inloggningsknappen kallas på i samma kod som inloggningsformulären, då detta passade bra med designen på hemsidan. Annan kod som visar information om användarens profil har också redigerats. En användare med LinkedIn-konto får inte längre se sitt användarnamn, då detta är en obegriplig textsträng. Formulär för att ändring av lösenord har också tagits bort när ett LinkedIn-konto används.

5.2.3 Säkerhet

I nuläget finns ett säkerhetskritiskt problem i lösningen, vilket är lagringen av de hemliga nycklarna. Problemet är oberoende av vilken OAuth-version som använts, då detta problem hade uppkommit oberoende av version. Hemligheterna är krypterade och sedan hårdkodade in i den nya klassen KeyFetcher, nyckel för att dekryptering ligger även denna i samma klass. Detta är inte önskvärt då utvecklare kan komma åt dessa nycklar. Det kräver dock direkt åtkomst till servern och/eller dess kod. Klassen KeyFetcher skapades för att stoppa utvecklare från att se nycklarna i klartext och för att enklare kunna bygga på säkerheten i efterhand.

I implementationen sparas ej *access token* i databasen, utan endast i sessionen. En session håller så länge användaren är inloggad på hemsidan, och förstörs när användaragenten stängs ner. Så en ny *access token* hämtas varje

gång användaren loggar in. Detta för att skydda sig mot databasintrång. Om någon lyckas kopiera delar eller hela databasen kan de inte komma åt LinkedIn-kontorna. Nackdelen med denna lösning är att det går lite långsammare att logga in, men inte tillräckligt för att inte genomföras.

6 Slutsats

Här beskrivs slutsatserna gentemot delfrågorna som tas upp i kapitel 1.3

Kapital 4.4 svarar på frågan ”Hur har de två versionerna av OAuth implementerats i LinkedIns servrar?”. Båda implementationerna har en god säkerhet. OAuth 1.0a har det i sin grund men LinkedIn har också lagt till variabeln *scope* för att förtydliga för användaren vilken information som delas ut. Säkerheten i OAuth 2.0 är mycket god, dock inte riktigt på samma nivå som deras OAuth 1.0a. Dokumentationen för denna version hade dock ett bättre upplägg och var mycket lättare att förstå.

Nästa fråga ”Vilken av de två versionerna av OAuth kan inom rimlig tid implementeras till PMCGs servrar?”. Svaret är båda, men om OAuth 1.0a valts hade det tagit längre tid. Detta beror på dokumentationen till OAuth 2.0 gjorde det mycket lättare att förstå hur implementationen skulle gå till väga. Detta tas upp i kapitel 4.2 och 4.4.

Kapitel 4.3 svarar på frågan ”Vilken säkerhetsnivå ligger de två versionerna av OAuth på?”. Protokollet OAuth 1.0a är ett mycket säkert protokoll, tack vare sina signaturer. Så länge nycklarna hålls hemliga gör inte ens en MITM attack mycket skada, då endast de uppfångade meddelanden kan användas, och endast en gång. Vill man ha ett säkert system skall denna version användas. Men de som skapar klienter kommer att få jobba lite extra och risken är att en del ger upp. OAuth 2.0 kan vara säkert och det kan vara osäkert. Problemet med denna version är att det lämnar mycket öppet, vilket gör att olika implementationer håller olika säkerhetsnivåer. Säkerhetsansvaret har alltså till dels delats ut till klienten genom användandet av SSL/TLS. Kommunikationen bryts inte när något inom SSL/TLS går fel, till exempel certifikatproblem, utan endast varningar skickas som lätt kan ignoreras av klienten eller servern. Skapas en säker serverimplementation räcker alltså inte det, utan serversidan får hoppas att klienterna sköter sitt. Den stora nackdelen är också versionens fördel. Den har många fler användningsområden, tack vara alla de olika flöden som kan användas. En server kan skräddarsy ramverket så det passar deras lösningar. OAuth 2.0 har även en brist om *access tokens* används som *bearer tokens*, vilket inte kräver någon typ av autentisering av klienten.

Sista frågan ”Hur ser framtiden ut för de två versionerna av OAuth?” svaras kort med att OAuth 1.0a antagligen kommer att försvinna. Alla nya serverimplementationer använder sig av OAuth 2.0 idag. Detta eftersom det är lättare för klienter att koppla upp sig och dels också tack vare den marknadsföringen som förs. Fårgan är bara hur lång tid det tar innan OAuth

1.0a försvinner. De som redan har skapat sig en sådan lösning har inte någon anledning att byta och kommer nog därför stanna kvar så länge de kan.

7 Utvecklingsmöjligheter

Nu är det fritt fram för PMCG att utnyttja LinkedIns REST API. Med detta kan mycket information hämtas om LinkedIn-användarna. Till exempel skulle information om vilka grupper en användare är med i på LinkedIn användas för att avgöra vilka rättigheter användaren har på PMCGs projektportal i olika projekt.

Just nu uppdateras inte en LinkedIn-användares uppgifter på PMCG om de ändras på LinkedIn. Detta på grund av LinkedIns användningsvillkor som dikterar att varje gång data hämtas och sparas undan måste användaren aktivt godkänna detta. En enkel lösning på detta problem kanske kan utvecklas. Kanske en checkbox bredvid inloggningsknappen "uppdatera mina uppgifter"?

Så som det ser ut just nu kan kopplingen mellan ett LinkedIn-konto och ett befintligt PMCG-konto endast ske första gången inloggning via LinkedIn sker. Denna funktion skulle kunna utökas så det är möjlig vid senare tillfällen.

Nycklarna är inte direkt synliga för ögat i prototypen, då de är krypterade. Den som dock kommer åt filen och har lite tid kan utan större ansträngning reda ut dessa. Säkerheten skulle kunna utökas på många sätt. Nycklarna skulle bland annat kunna ligga i en separat fil utanför serverstrukturen, för att minska risken att den upptäcks. Även här i skulle de kunna vara krypterade som de är nu, eller ännu hårdare. För att ytterligare öka säkerheten kan stränga rättigheter implementeras så filen öppnas från rätt plats.

8 Referenser

8.1 Källkritik

Hueniverse är en blogghemsida skapad av Evan Hammer-Lahav som är en av de som varit med och utvecklat båda versionerna. Det var även Hammer-Lahav som skrev den första specifikationen. Han har god förståelse för ämnet och är därför en god referens. Det är dock bra att veta att han är en stor motståndare till ramverket.

Homakovs bloggsida visar bland annat på många attacker som han gör mot olika OAuth lösningar och går noggrant igenom hur dessa attacker skall genomföras. Så man kan själv testa alla hans teorier.

8.2 Källor

1. Hammer-Lahav, E., April 2010, *The OAuth 1.0 Protocol*, IETF, http://tools.ietf.org/pdf/rfc5849.pdf

2. Hardt, D., Oktober 2012, *The OAuth 2.0 Authorization Framework*, IETF, http://tools.ietf.org/pdf/rfc6749.pdf

3. Lodderstedt, T., Januari 2013, *OAuth 2.0 Threat Model and Security Considerations*, IETF, http://tools.ietf.org/html/rfc6819.html

4. Boyd, R., februari 2012, Getting started with Oauth 2.0.O'REILLY, ISBN 978-1-449-31160-5

5. http://hueniverse.com/2010/09/OAuth-bearer-tokens-are-a-terrible-idea/, september 2010, Evan Hammer-Lahavs hemsida.

6. http://hueniverse.com/2010/09/OAuth-2-0-without-signatures-is-bad-for-the-web/ september 2010, Evan Hammer-Lahavs hemsida.

7. http://hueniverse.com/2009/04/explaining-the-OAuth-session-fixation-attack/, april 2009, Evan Hammer-Lahavs hemsida

8.
http://hueniverse.com/2011/06/OAuth-2-0-redirection-uri-validation/, Juni 2011, Evan Hammer-Lahavs hemsida.

9.
http://hueniverse.com/2012/11/fuckoauth-realtimeconf/, november 2012, Evan Hammer-Lahavs hemsida.

10.
Krawczyk, H. m.fl., februari 1997, HMAC: Keyed-Hashing for Message Authentication, Network Working Group, http://www.faqs.org/rfcs/rfc2104.html

11.
http://csrc.nist.gov/groups/ST/hash/statement.html, april 2006, NISTs hemsida

12.
http://homakov.blogspot.co.uk/2013/02/hacking-facebook-with-OAuth2-and-chrome.html, februari 2013, Egor Homakovs säkerhetsblogg

13.
LinkedIn.com, LinkedIn APIs Terms of Use, http://developer.linkedin.com/documents/linkedin-apis-terms-use

14.
LinkedIn.com, LinkedIn documentation for OAuth 2.0, http://developer.LinkedIn.com/documents/authentication

15.
OAuth.net, OAuths officiella hemsida, http://oauth.net/

9 Akronymer

OAuth	**O**pen **Auth**orization
SVN	**S**ub**v**versio**n**
API	**A**pplication **P**rogramming **I**nterface
REST	**Re**presentational **S**tate **T**ransfer
URI	**U**niform **R**esource **I**dentifier
URL	**U**niform **R**esource **L**ocator
CMS	**C**ontent **M**anagement **S**ystem
SSL	**S**ecure **S**ocket **L**ayer
TLS	**T**ransport **L**ayer **S**ecurity
XSS	**C**r**oss-S**ite **S**cripting
MITM	**M**an **I**n **T**he **M**iddle
CSRF	**C**ross-**S**ite **R**equest **F**orgery
IETF	**I**nternet **E**ngineering **T**ask **F**orce
NIST	**B**ational **I**nstitute of **S**tandards and **T**echnology