# HYBRID BIOMETRIC AUTHENTICATION PROTOCOL DESIGN USING PROVERIF

*Thesis submitted in partial fulfillment of the requirements for the award of degree of*

**Master of Engineering**

in

**Computer Science and Engineering**

*Submitted By*


**NEHA ARORA**

**( 801132019)**


Under the supervision of:

**MR. SUMIT MIGLANI**

Asstt.Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA – 147004


June 2013

# TABLE OF CONTENTS

**CERTIFICATE…………………………………………………………………….....i**
**ACKNOWLEDGEMENTS…………………………………………………….....ii**
**ABSTRACT……………………………………………………………………......iii**
**TABLE OF CONTENTS……………………………………………………...iv**
**LIST OF FIGURES…………………………………………………………....vi**
**LIST OF TABLES………………………………………………………………vii**

**CHAPTER 1 INTRODUCTION**…………………………………………………..**1**

1.1 Double redirection protocols…………………………………………1

1.2 Research Contribution………………………………………………..3

1.3 Objective…………………………………………………………....4

**CHAPTER 2 LITERATURE REVIEW**……………………………………………..**5**

2.1 Basic Knowledge of Protocol……………………………………………..5

2.2 Security Aspects……………………………………………………………..6

2.3Major security Threats related to Authentication Protocols………………7

2.3.1 Web Based Attacks………………………………………………..7

2.3.2 Social CSRF Attacks………………………………………………8

2.4Social Sign On and Sharing…………………………………………………...9

**CHAPTER 3 PROBLEM STATEMENT**……………………………………………..**11**

3.1 Research Question…………………………………………………….11

3.2 OAUTH Protocol…………………………………………………...12

3.2.1DelegatedAuthentication………………………………………..12

3.2.2Scenario…………………………………………………………13

3.2.3Versions…………………………………………………………14

3.2.4Necessity……………………………………………………..15

3.2.5Limitations……………………………………………………16

3.4 OAUTH2.0 ……………………………………………………………17

**CHAPTER 4 IMPLIMENTATION**…………………………………………………...**18**

4.1 Methodology……………………………………………………………18

4.2 Cryptographic Protocol Verifier:Proverif………………………………18

4.2.1Modelling Protocols using Proverif……………………………...19

4.2.2 Webspi Library ……………..……………………………………...20

4.2.3Cryptography……………………………………………………..20

4.3Digital Persona Software…………………………………………………20

4.3.1Working…………………………………………………………..21

4.4Facebook Graph API Explorer………………………………………….21

4.5OAUTH2.0 in Proverif………………………………………………….23

4.5.1User Agent flow…………………………………………………23

4.5.2Authorisation Code Flow………………………………………..24

4.5.3Addition Protocol Parameters…………………………………..26

4.5.4Threat Model…………………………………………………….27

4.6Security Goals…………………………………………………………..27

4.7Modelling Principals……………………………………………………29

4.8Attacker Model………………………………………………………….31

4.8.1Attacker API……………………………………………………31

4.9Sumarisation…………………………………………………………….32

4.10Hybrid Authentication Protocol……………………………………….33

4.10.1Showase Of Attack on OAUTH2.0 Protocol………………….37

4.10.2Implementing Hybrid Biometric authentication Protocol…….39

4.10.3Output…………………………………………………………41

**CHAPTER 5 RESULTS AND ANALYSIS…**……………………………………………...**42**

5.1 Analysis of Protocol…………………………………………………42

5.1.1Privacy of data……………………………………………………..42

5.1.2Liveness…………………………………………………………42

5.1.3Intensional Authentication………………………………………42

5.2Limitation……………………………………………………………..43

**CHAPTER 6 CONCLUSION AND FUTURE**……………………………………..44

6.1Conclusion…………………………………………………………..44

6.2Future scope………………………………………………………...44

**REFERENCES**

**PAPER PUBLISHED**

**ACRONYMS**

# LIST OF FIGURES

Figure 1.1: Social Sign On 1

Figure 2.1: Login on Yahoo with Gmail 9

Figure 3.1 How Does OUTH 2 Works 13

Figure 3.2 Social Login on twitter by Bit.ly 14

Figure 4.1: Identification of fingerprints 21

Figure 4.2: Screen shot of Access Token Generation 22

Figure 4.3: Using Access Token to create events 23

Figure 4.4: User Agent Flow 24

Figure 4.8:Screenshot of Hybrid Authentication Protocol 41

# LIST OF TABLES

Table 3.1:Difference between OAUTH and OAUTH2.0 18

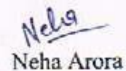
Table 4.1:Modelling Pricipals 32

Table 4.2:Modelling Principals(Hybrid Authentication Protocol) 36

# CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Hybrid authentication Protocol Design Using Proverif*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Computer Science and Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr.Sumit Miglani* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

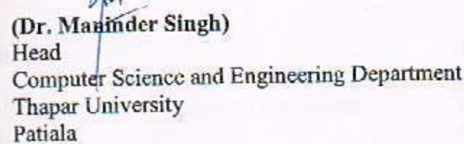


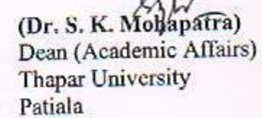
Neha Arora

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Mr.Sumit Miglani
Asstt. Professor
CSED

Countersigned by





**(Dr. Maninder Singh)**
Head
Computer Science and Engineering Department
Thapar University
Patiala

**(Dr. S. K. Mohapatra)**
Dean (Academic Affairs)
Thapar University
Patiala

# Acknowledgements

It is a great pleasure to have an opportunity to thanks valuable beings for their continuous support and inspiration throughout the thesis work.

I would like to extend my gratitude towards Dr. Maninder Singh, HOD of Computer Science and Engineering Department, Thapar University for all the guidance and great knowledge he shared during our course. The abundance of knowledge he has always satisfied our queries at every point.

Thanks to Mr. Sumit Miglani, My guide for his contribution for timely reviews and suggestions in completing the thesis. Every time he provided the needed support and guidance.

At last but not the least, a heartiest thanks to all my family and friends for being there every time I needed them.

# ABSTRACT

The thesis presents verification of OAUTH and OAUTH2 Protocol. ProVerif is used as the verification tool for verifying and analysing the protocols. The protocol are analysed in ProVerif model. Various attacks to the protocols are generated in order to verify whether the protocols hold their intended properties.We have selected 2 protocols and proposed a Hybrid OAUTH biometric authentication protocol for social login. Each of which has different intended purposes and properties. The first protocol is generic authentication highlighting the deficiencies. This protocol provides three properties of the protocol: effectiveness, correctness, and privacy of data.However,it suffers from a large count of vulnerabilities like CSRF attack,phishing attacks and so on. The second protocol is a new evolution of previous protocol with major changes in data flow and and security aspects.Though,it overcomes many of the vulnerabilities but still the security of data was questionable. The two properties of the protocol are verified: privacy of bio-metric data and intensional authentication. Hence, third protocol was designed so that the intensional authentication property can be verified. The protocol promises three intended properties: privacy of the biometric data, liveness of biometric data used as a salt in token secret of MAC token used in OAUTH2 and intensional authentication. The protocol is illustrated in detail and desirable properties of the protocol are verified.

# CHAPTER 1:INTRODUCTION

User authentication is a method to verify the user's identity.The process seems to be a child's play when asked to a layman.To a normal person it may appear as if a login page appears.One enters his username and password and accesses the account.But have we ever pondered on the fact what is the actual process.It is all the the game of RP(Resource Providers)such as facebook,google API,to name a few and IDP().And the process further becomes complicated when we social login."Social Login" is logging on to a particular website through credentials used in another website like logging yahoo through facebook.The protocols employed in such process are know as Double Redirection Protocols

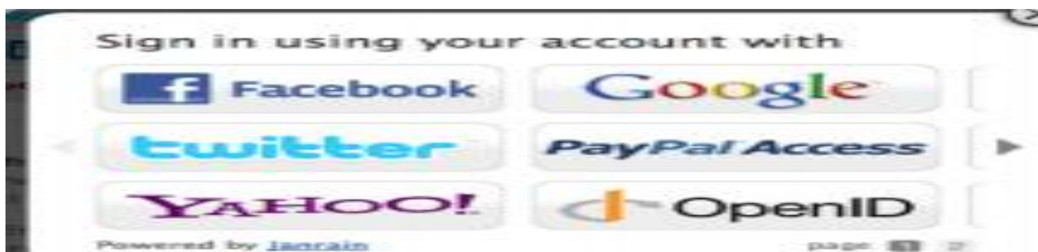



Figure 1.1:Social sign on

## 1.1 DOUBLE REDIRECTION PROTOCOLS

A double-redirection protocol is a security protocol where an application redirects the user's browser to a third-party that interacts with the user before redirecting the user back to the application. The third party identifies the application to the user, authenticates the user, and asks for permission to identify the user to the application and grant the application access to resources and services on behalf of the user.The application may be a traditional Web application running on a Web server; a rich application running within the browser and implemented, e.g., in JavaScript, Flex or Silverlight; or a native application running on a desktop or mobile platform. The third party may be, for example, a social site such as Facebook, a microblogging site such as Twitter, a photo-sharing site such as Flickr, a multipurpose Web site such as Yahoo, Google or Windows Live, a dedicated Web identity provider, a cloud services identity provider, or an enterprise authorization server. The third party may grant the application access to resources and services that it controls, or to resources and

services controlled by fourth parties.The prototypical double-redirection protocols are OpenID  and OAuth , but there are others,including combinations of OpenID and OAuth  and the Web Browser Single-Sign-On (SSO)Profile of SAML .

OpenID and OAuth are described respectively as an authentication protocol and an authorization protocol, but the distinction between authentication and authorization is not sharp, and the two protocols have overlapping functionalities and use cases. Double-redirection protocols are becoming key elements of the Web ecosystem, as they are used for social login . Social login adds to the advantages of Web SSO the compelling advantage for the application of gaining access to the user's social context.Prior work on the security of double-redirection protocols includes security-consideration sections of protocol specifications, criticism of OpenID , criticism of OAuth 2.0 by the very editor of the specification , and discussions of the OAuth 2.0 specification, which is still under development, on the mailing list of the IETF OAuth working group .

## 1.2 RESEARCH CONTRIBUTION

We model various configurations of the  protocol in the applied picalculus and analyze them using ProVerif . Our models rely on a generic library, WebSpi, that defines the basic components (users, browsers, HTTP servers) needed to model web applications and their security policies. In order to express realistic security goals for a web application, we show how to encode distributed authorization policies in the style of  in ProVerif. The library also defines an operational web attacker model so that attacks discovered by ProVerif can be mapped to concrete website actions closely corresponding to the actual PHP implementation of an exploit. The model developer can fine-tune the analysis by enabling and disabling different classes of attacks. The effectiveness of our approach is testifed by the discovery of several previously unknown vulnerabilities involving some of the most popular websites, including Facebook, Yahoo, and Twitter. We reported these problems and helped overcome them.

A double-redirection protocol consists of:

1. The following core steps, in sequence:

(a) (First redirection.) The application redirects the browser to a third party user-interaction endpoint, requesting user data which may include identity data, and/or permission to access resources or services on behalf of the user.

(b) The third party identifies the application to the user, describes to the user the request made by the application, asks for permission to fully or partially satisfy the request, and authenticates the user.

(c) (Second redirection.) If permission was granted in step 1(b), the third party redirects the browser to a callback endpoint of the application, sending a user identifier and/or user data and/or a token; the token may be an access token that provides access to requested resources or services, or a provisional token that the application exchanges for an access token.

(d) The application verifies and/or obtains data. If the response conveyed by the second redirection in step 1(c) included a user identifier and/or user data, the application may verify the origin of the data by verifying a signature included in the response. If it included an access token (or a provisional token that was exchanged for an access token) and the protocol is used for social login, the application uses the access token to obtain a user identifier and/or user data from the third party.(e) If the response conveyed by the second redirection in step 1(c) included an access token (or a provisional token that was exchanged for an access token), the application uses the access token to request access to resources and services on behalf of the user at access endpoints controlled by the third party or by fourth parties.

2. An application authentication step, which is placed at different positions in the above sequence by different protocols, and is omitted by some protocols

**In this thesis we add the following contributions to the prior work:**

1. We describe a recently-discovered attack against double-redirection protocols

2. We specify conditions that we believe to be sufficient for achieving security with a double-redirection protocols

3. We propose new building blocks for designing secure double-redirection protocols, including:

(a) A new method for authenticating the application without ad-hoc signatures;

(b) Methods for authenticating rich applications and native applications without exposing application secrets to the user by making changes in message passing methods.

(c) New methods for passing access tokens to rich applications and native applications using biometrics as a salt.

Hence,we developed a new hybrid authentication protocol based on these propositions.

The tools used are

1.Proverif

2.Digital Persona

3.Facbook Graph API Explorer

## 1.3OBJECTIVE

The main objective of this thesis is to analyse the protocols on Proverif.Demonstrate the attacks and propose a new protocol free from vulnerabilities and analyse it. ProVerif is an automatic cryptographic protocol verifier, in the formal model (so called Dolev-Yao model). This protocol verifier is based on a representation of the protocol by Horn clauses. Its main features are:

- It can handle many different cryptographic primitives, including shared- and public-key cryptography (encryption and signatures), hash functions, and Diffie-Hellman key agreements, specified both as rewrite rules or as equations.
- It can handle an unbounded number of sessions of the protocol (even in parallel) and an unbounded message space. This result has been obtained thanks to some well-chosen approximations. This means that the verifier can give false attacks, but if it claims that the protocol satisfies some property, then the property is actually satisfied.

## 1.4 PUBLICATIONS RESULTING FROM THIS THESIS

# CHAPTER 2:LITERATURE REVIEW

Here, a summary of related research is presented. Topics covered include basic knowledge of authentication, biometric data protection, veri-fication of security protocols, authentication protocols,access token and trusted platforms . In addition, Dolev-Yao style attackers,phising attacks ,CSRF attacks and possible problems with prior protocols are included. Some basic definitions are also presented in this chapter.

## 2.1BASIC KNOWLEDGE OF REDIRECTION PROTOCOL

A double-redirection protocol consists of:

1. The following core steps, in sequence:

(a) (First redirection.) The application redirects the browser to a third party user-interaction endpoint, requesting user data which may include identity data, and/or permission to access resources or services on behalf of the user.

(b) The third party identifies the application to the user, describes to the user the request made by the application, asks for permission to fully or partially satisfy the request, and authenticates the user.

(c) (Second redirection.) If permission was granted in step 1(b), the third party redirects the browser to a callback endpoint of the application, sending a user identifier and/or user data and/or a token; the token may be an access token that provides access to requested resources or services, or a provisional token that the application exchanges for an access token.

(d) The application verifies and/or obtains data. If the response conveyed by the second redirection in step 1(c) included a user identifier and/or user data, the application may verify the origin of the data by verifying a signature included in the response. If it included an access token (or a provisional token that was exchanged for an access token) and the protocol is used for social login, the application uses the access token to obtain a user identifier and/or user data from the third party.(e) If the response conveyed by the second redirection in step 1(c) included an access token (or

a provisional token that was exchanged for an access token), the application uses the access token to request access to resources and services on behalf of the user at access endpoints controlled by the third party or by fourth parties.

2. An application authentication step, which is placed at different positions in the above sequence by different protocols, and is omitted by some protocols

## 2.2 SECURITY ASPECTS

Any communication happening over network requires having security. So, it needs to happen in the two links only in complete isolation from other systems though at the same are also part of the same network. This is what that formed the basis for securing the communication. For this securing the computer is done to secure the information and valuable assets in the computer. This could also be called the hardening of the system.

So, the whole concept of network security is based to have information security. The major goals of this information security are:

- **Confidentiality:** It means the protecting the information that is present in a system from unauthorized people. Such as information regarding customer credit card, information of patients in hospitals or information related to employees of an organization. If information of such level of confidentiality is not secured, the company or the organization involved will probably lose its reputation and goodwill in the market.
- **Integrity:** It indicates that information available in an organization should be complete and whole. It should not be changed by any unauthorized person. Any kind of intentional or unintentional alterations of the information will lead to damaging and making of information unreliable. A good example of such a case would be account information in a Bank. If anything is done with banking information, it is devastating and the Bank will eventually lose its customers and business. In fact, there is also a chance of facing a lawsuit also.
- **Authenticity:** It is the identification and assurance of the origin the data. It ensures that the access to the data should be to trusted people only. Valid usernames and passwords should be given and these should be kept confidential.

- **Availability:** It says the information requested or required by the authorized users should be available, always. For example, suppose a company met with a natural calamity and it has lost its computers and all the important data. In such cases, the affected company should be able to install new computers and recover its data from backups. Suppose, the company had not had any proper backup plans, they would definitely be not able to recover their data and resume its operations.

## 2.3MAJOR SECURITY THREATS RELATED TO AUTHENTICATION PROTOCOLS

### 2.3.1 WEB BASED ATTACKS

Network attacks are well understood, and can be mitigated by the systematic use of HTTPS ,or more sophisticated cryptographic mechanisms. Many websites, such as Facebook (at the time of writing), do not even seek to protect against network attackers for normal browsing, allowing users to access their data over HTTP. They are more concerned about website- and browserbased attacks, such as Cross-Site Scripting (XSS), SQL Injection, Cross-Site Request Forgery (CSRF) and Open Redirectors [1]. For example, various avors of CSRF are common on the web. When a user logs into a website, the server typically generates a fresh, unguessable, session identiFer and returns it to the browser as a cookie. All subsequent requests from the browser to the website include this cookie, so that the website associates the new request with the logged-in session. However, if the website relies only on this cookie to authorize security-sensitive operations on behalf of the user, it is vulnerable to CSRF.

A malicious website may fool the user's browser into sending a (cross-site) request to the vulnerable website (by using JavaScript, HTTP redirect, or by inviting the user to click on a link). The browser will then automatically forward the user's session cookie with this forged request, implicitly authorizing it without the knowledge of the user, and potentially compromising her security. A special case is called login CSRF: when a website's login form itself has a CSRF vulnerability, a malicious website can fool a user's browser into silently logging In to the website under the attacker's credentials, so that future user actions are credited to the attacker's account.

A typical countermeasure for CSRF is to require every security-sensitive request to include a session-speci c nonce that would be difficult for a malicious website to forge. This nonce can be embedded in the target URL or within a hidden form  eld. However, such mechanisms are still not widely deployed and CSRF attacks remain prevalent on the web, even on respected websites.

### 2.3.2 SOCIAL CSRF ATTACKS

Cross-Site Request Forgery (CSRF) is an attack that tricks the victim into loading a page that contains a malicious request. It is malicious in the sense that it inherits the identity and privileges of the victim to perform an undesired function on the victim's behalf, like change the victim's e-mail address, home address, or password, or purchase something. CSRF attacks generally target functions that cause a state change on the server but can also be used to access sensitive data.

For most sites, browsers will automatically include with such requests any credentials associated with the site, such as the user's session cookie, basic auth credentials, IP address, Windows domain credentials, etc. Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish this from a legitimate user request.

In this way, the attacker can make the victim perform actions that they didn't intend to, such as logout, purchase item, change account information, retrieve account information, or any other function provided by the vulnerable website.

Sometimes, it is possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called Stored CSRF flaws. This can be accomplished by simply storing an IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attack can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already.

The above attacks have been discovered by studying the concept of social login and sharing

## 2.4 SOCIAL SIGN ON AND SOCIAL SHARING

Social sign-on (or social login) is the use of a social network to login to a third-party website, without having to register at the website. It is a service provided by many social networks and authentication servers, using protocols such as OpenID (e.g. Google) and OAuth (e.g.Facebook). For clarity, we henceforth adopt OAuth terminology: a user who owns some data is called a resource owner, a website that holds user data and o users API access to it is called a resource server, and a third party that wishes to access this data is called a client or an app

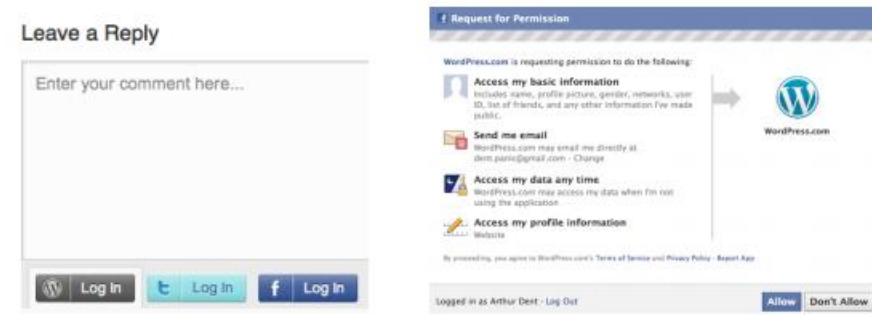



Figure 2: (L) Log in with Facebook on Wordpress; (R) Facebook requires authorization.

**Motivating Example:**

Consider WordPress.com, a website that hosts hundreds of thousands of active blogs with millions of visitors every day. A visitor may comment on a blog post only after authenticating herself as a WordPress, Facebook, or Twitter user (Figure 1-L). When a visitor Alice clicks on \Log in with Facebook", an authorization protocol is set into motion where Alice is the resource owner, Facebook the resource server, and WordPress the client. Alice's browser is redirected to Facebook.com which pops up a window asking to allow WordPress.com to access her Facebook profile (Figure 1-R). WordPress.com would like access to Alice's basic information, in particular her name and email address, as proof of identity. If Alice authorizes this access, she is sent back to WordPress.com with an API access token that lets WordPress.com read her email address from Facebook and log her in, completing the social sign-on protocol. All subsequent actions that Alice performs at WordPress.com, such as commenting on a blog, are associated with her Facebook identity. Some client websites also implement social sharing: reading and writing data on the resource owner's social network. For example, on CitySearch.com, a guide with restaurant and hotel recommendations, any review or comment written by a logged-in Facebook user is instantly cross-posted on her profile feed (`Wall') and shared with all her friends. Some websites go even

further: Yahoo.com acts as both client and resource server to provide deep social integration where the user's social information  ows both ways, and may be used to enhance her experience on a variety of online services, such as web search and email.

## 2.5 PERFORMANCE BASED WORK

Analysis of latest open redirection protocol OAUTH2 has been done using an open source tool and then some improvements are made in it thereby constructing a new protocol hybrid authentication protocol.Comparison of three protocols is done and which one is better is shown.

# CHAPTER 3:PROBLEM STATEMENT

## 3.1 IS SOCIAL LOGIN STILL SAFE??????????

Though OAUTH2.0 has been introduced to overcome the limitations of OAUTH bt OUTH2.0 did not prove to be a significant improvement and hence did not justify the role of an authentication protocol therefore it is just referred to as a framework rather than a protocol.It suffered from problems like unbounded token,lack of security and many more which have been depicted in next chapter using proverif and a new protocol has been devised.Studying the above two protocols we have come to know that the loophole lies in the access token,which is studied in detail in next chapter.Analysis of OAUTH2 has been done using proverif and a CSRF attack has been implemented.

## 3.2OAUTH PROTOCOL

The OAuth protocol was originally created by a small community of web developers from a variety of websites and other Internet services who wanted to solve the common problem of enabling delegated access to protected resources. The resulting OAuth protocol was stabilized a version 1.0 in October 2007, and revised in June 2009 (Revision A) In the traditional client-server authentication model, the client uses its credentials to access its resources hosted by the server. With the increasing use of distributed web services and cloud computing, third-party applications require access to these server- hosted resources. OAuth introduces a third role to the traditional client-server authentication model: the resource owner. In the OAuth model, the client (which is not the resource owner, but is acting on its behalf) requests access to resources controlled by the resource owner, but hosted by the server. In addition, OAuth allows the server to verify not only the resource owner authorization, but also the identity of the client making the request. OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different

client or an end- user). It also provides a process for end-users to authorize third party access to their server resources without sharing their credentials (typically, a username and password pair), using user- agent redirections. For example, a web user (resource owner) can grant a printing service (client) access to her private photos stored at a photo sharing service (server), without sharing her username and password with the printing service. Instead, she authenticates directly with the photo sharing service which issues the printing service delegation-specific credentials.In nutshell, OAuth stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation, and is the right solution for securing open platforms.

The developers of OAuth set out to solve the problem that services and passwords don't mix well as you start to combine apps and mash them up. Imagine, in today's environment of web APIs and mobile apps, if every web site that used an API from another web site had to share and store that web site password. Soon you'd have the proliferation of your passwords all over the Internet with every service you've used from Facebook, to Twitter, to Skype, to your bank account

### 3.2.1THE VALET KEY METAPHOR

The valet key metaphor OAuth supports this "delegated authentication" between web apps using a security token called an "access token." Rather than relying on a single password as the master key for every app that accesses an API, OAuth uses this token. An OAuth token gives one app access to one API on behalf of one user.Eran Hammer-Lahav, a spec author for OAuth, compares the token to a valet key. This is an apt metaphor.For most people, their car is one of their most valuable possessions, valued in tens of thousands of dollars. They are convenient places to leave our other valuables like computers and clothing. Yet we are sometimes required to give them to a parking attendant or valet whom we've not met before. A valet key solves the problem – it's an access token with limited rights that can operate the vehicle but not grant access to the trunk or glove box.

**How does OAUTH work??????**

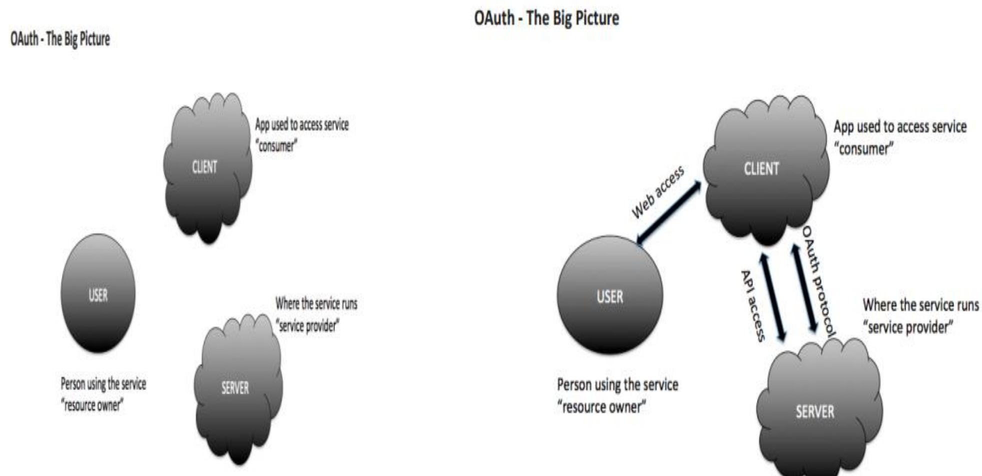



Figure 3.1:How OAUTH2 works

Simply, there are three entities (legs) to consider for an OAuth scenario:

1. The user of a service – let's call him Bob

2.A client (a Web app, a mobile app, or a server) – let's take the URL-shortening service bit.ly as an example client

3. The server (where the service runs) – let's take Twitter as an example

## 3.2.2 SCENARIO

How does our user Bob interact with Twitter through his bit.ly account?

1. Bob visits bit.ly on the web, which uses a service provided by Twitter. Bob already has logins for bit.ly and Twitter.

2. Behind the scenes, bit.ly uses it's OAuth credentials to begin the authentication process for Bob.

Bit.ly redirects Bob temporarily to twitter.com to log in. (bit.ly never sees Bob's Twitter password).The page hosted by Twitter that asks if an application can access Twitter on your behalf is probably familiar to most of us by now. Note that Twitter shows Bob what rights the app is asking for, and importantly what the app will not be able to do – in this case, see that bit.ly cannot access Bob's direct messages or see his Twitter password.

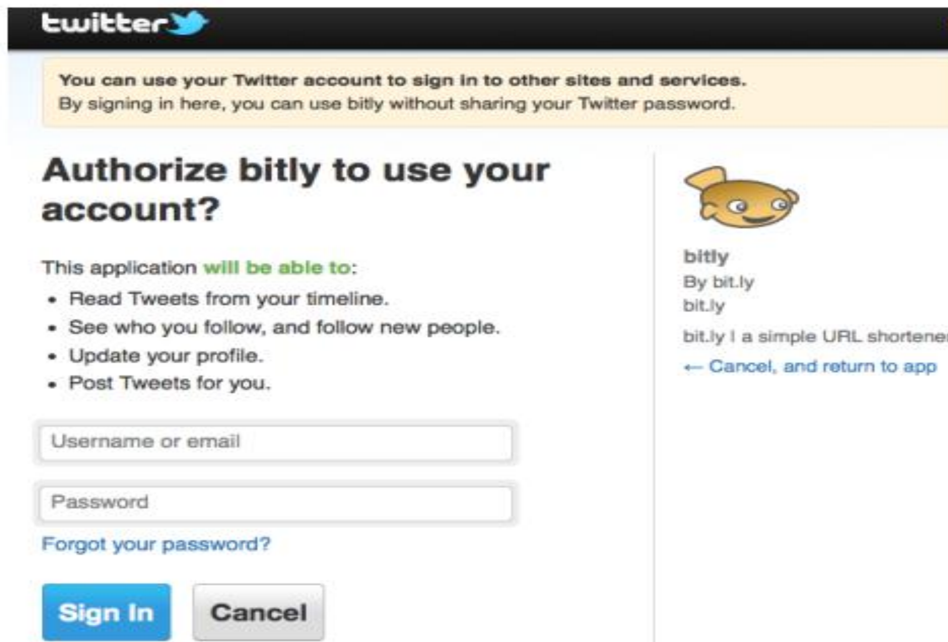



Figure 3.2 Social Login of twitter through bit.ly

3. If this sign in is successful, bit.ly uses its own OAuth credentials (token) to retrieve credentials for Bob (that's the valet key that allows bit.ly to use Twitter on Bob's behalf).

4. Bit.ly stores Bob's credentials along with Bob's account. They allow him to use bit.ly and only bit.ly to access Twitter.

### 3.2.3 VERSIONS

**OAuth 1.0** The first "production" version of OAuth 1.0 didn't actually do authentication delegation correctly and as a result, the spec itself had to be patched. No one should be using OAuth 1.0 now.

**OAuth 1.0a** Not an IETF standard but stable and well understood.Uses a form of a cryoptographic hash code to sign each API request and exchange credentials between client and server. This means that you get API security without SSL (not only is the password never exchanged but the OAuth token is never exchanged – it's encrypted inside the signature). But coding the signature right is tricky.OAuth 2.0 Actively under development in the IETF.

**OAuth 2.0** is quite stable now – core flows are unlikely to change. It supports a 'bearer token', which is easier to code than the signature but requires SSL. (However,

most APIs should be using SSL by default anyway.) Optional specs support signatures, SAML, and so on, but those specs are not yet stable

### 3.2.4 NECESSITY

OAuth has become the best practice and is essential for enabling secure user access and providing smooth user experiences. We strongly recommend OAuth for API providers when they are exposing APIs for web or mobile apps, but there are a couple of scenarios in which OAuth might be overkill or not the best solution.

**For server-to-server APIs** - APIs designed to be used only by a small number of servers –OAuth is overkill. Having a separate set of authentication credentials for each app is a nice feature of OAuth, but for server-to-server use, the need to log in securely using a browser, or to implement other steps in the OAuth "dance," gets in the way.Instead, using a simple security standard like HTTP Basic authentication and assigning a unique password to each app is sufficient. Two-way SSL is another good, albeit cumbersome approach that has the advantage of stronger, more traceable authentication.However, think ahead! Are those APIs really only going to be used by servers forever? In the future perhaps they'll be used by web apps or mobile clients, or the number of servers will grow far beyond what you first imagined. Furthermore, OAuth 2.0 has a number of ways for servers to securely get tokens that don't require a browser-based login. OAuth might seem like overkill today, but emerge as totally critical a few months down the road.

**Use SSL for everything sensitive**. Unless your API exclusively has open and non-sensitive data, support SSL and consider enforcing it by redirecting any API traffic on the non-SSL port to the SSL port. It makes other authentication schemes more secure, and keeps your user's private API data from prying eyes—and it's not all that hard to do.

**Use API keys only for non-sensitive, read-only data. If you have a public API—** which exposes data you'd make public on the Internet anyway–consider issuing non-sensitive API keys. These are easy to implement and still give you a way to identify applications and developers. Armed with an API key, you have the option of establishing a quota for your API, or at least monitoring usage by application. Without one, the only way to track usage is through IP addresses, which are not reliable. For example, the Yahoo Maps Geocoding API issues API keys so it can track

its users and establish a quota, but the data that it returns is not sensitive, so it's not critical to keep the key secret. ("Two-legged OAuth" is another way to do this that protects the API key on the network without requiring SSL.)

**Sanitize incoming and outgoing data**. This will prevent malicious code or content from entering your system or being executed by clients that read your data. This practice should be employed for all APIs, but may be especially important for writable ones. For example, for a writable API, you do not want to allow insertion of JavaScript that could be used for cross-site scripting attacks when users visit your website. For example, if a parameter value is known to be numeric, it should be validated as numeric before being passed. Again, this practice should be applied for all APIs, but may be more needed for writable APIs.

### 3.2.5 LIMITATIONS

In OAuth, the shared secret depends on the signature method used. In the PLAINTEXT and HMAC-SHA1 methods, the shared secret is the combination of the Consumer Secret and Token Secret. In the RSA-SHA1 method, the Consumer Private Key is used exclusively to sign requests and serves as the asymmetric shared secret. The way asymmetric key-pairs work, is that each side — the Consumer and Service Provider — use a one key to sign the request and another key to verify the request. The keys — Private Key for Consumer and Public Key for the Service Provider — must match, and only the right pair can successful sign and verify the request. The advantage of using asymmetric shared secrets is that even the Service Provider does not have access to the Consumer's Private Key which reduces the likelihood of the secret being leaked.

However, since the RSA-SHA1 method does not use the Token Secret (it doesn't use the Consumer Secret either but that is adequately replaced by the Consumer Private Key), the Private Key is the only protection against attacks and if compromised, puts all Tokens at risk. This is not the case with the other methods where one compromised Token Secret (or even Consumer Secret) does not allow access to other resources protected by other Tokens (and their Secrets).

When implementing OAuth, it is critical to understand the limitations of shared secrets, symmetric or asymmetric. The Consumer Secret (or Private Key) is used to

verify the identity of the Consumer by the Service Provider. In case of a web-based Consumer such as web server, it is relatively easy to keep the Consumer Secret (or Private Key) safe. However, when the Consumer is a desktop application, a mobile application, or any other client-side software such as browser applets (Flash, Java, Silverlight) and scripts (JavaScript), the Consumer credentials must be included in each copy of the application. This means the Consumer Secret (or Private Key) must be distributed with the application, which inheritably compromises them.

This does not prevent using OAuth within such application, but it does limit the amount of trust Service Provider can have in such public secrets. Since the secrets cannot be trusted, Service Provider must treat such application as unknown entities and use the Consumer identity only for activities that do not require any level of trust, such as collecting statistics about applications. Some Service Provider may opt to ban such application or offer different protocols or extensions. However, at this point there is no known simple solution to this limitation.

It is important to note, that even though the Consumer credentials are leaked in such application, the User credentials (Token and Secret) are specific to each instance of the Consumer which protects their security properties. This of course greatly depends on the Consumer implementation and how it stores Token information on the client side.

This is somewhat different in a 2-legged scenario since the Consumer credentials are basically just a username and password and there is no Token or Token Secret. When OAuth is used as a direct replacement for HTTP 'Basic' making the Consumer and User the same entity, the application can be written to prompt Users to enter their credentials, hence removing the need to hard-code them into the application itself. Using OAuth for simple sign-in has the same user experience as HTTP 'Basic', but when used over an insecure channel OAuth provides much greater security.

## 3.3 OAUTH 2.0

OAuth 2.0 is designed to address these shortcomings. The protocol specification defines several different flows or protocol configurations, two of which directly apply to website applications. The protocol itself requires no cryptographic mechanisms whatsoever and instead relies on transport layer security (HTTPS). Hence, it claims to

be lightweight and flexible, and has fast emerged as the API authorization protocol of choice, supported by Microsoft, Google and Facebook, among others. We in next chapter describe the two website flows of OAuth 2.0, their security goals, and their typical implementations

Table 3.1Difference between OAUTH and OAUTH2

| OAUTH | OAUTH2 |
| --- | --- |
| No support for browser based apps | Introduced more flows to support browser based applications |
| It requires client applications to have cryptography | It no longer requires that |
| Signatures are more complicated | Signatures are less complicated |
| Access tokens are not refreshed | Access tokens are refreshed |
| | |

# CHAPTER 4 :IMPLEMENTATION

## 4.1 METHODOLOGY

There are various methods of analysing the protocols bt we have used proverif because it automatically models an attacker compliant with dolev-yao.we need not explicitly model the attacker.

| | |
|---|---|
| P,Q,R | Processes |
| 0 | null process |
| P\|Q | parallel composition |
| new n;P | name restriction |
| new x;P | variable restriction |
| if M=N then P else Q | condition |
| event x;P | event launch |
| let x=M in P | replace x with term M in process P |
| in(M,x);P | message input |
| out(M,x);P | message output |

For Biometric data we have used digital persona Software for fingerprint capture and access token (discussed in section 4.4) is generated using facebook graph API explorer

## 4.2 CRYPTOGRAPHIC PROTOCOL VERIFIER:PROVERIF

**ProVerif** is a software tool for automated reasoning about the security properties found in cryptographic protocols. The tool has been developed by Bruno Blanchet.

Support is provided for cryptographic primitives including: symmetric & asymmetric cryptography; digital signatures; hash functions; bit-commitment; and signature proofs of knowledge. The tool is capable of evaluating reachability properties, correspondence assertions and observational equivalence. These reasoning capabilities are particularly useful to the computer security domain since they permit the analysis of secrecy and authentication properties. Emerging properties such as

privacy, traceability and verifiability can also be considered. Protocol analysis is considered with respect to an unbounded number of sessions and an unbounded message space. The tool is capable of attack reconstruction: when a property cannot be proved, an execution trace which falsifies the desired property is constructed

### 4.2.1MODELLING PROTOCOLS USING PROVERIF

A ProVerif model of a protocol, written in the tool's input language (the typed pi calculus), can be divided into three parts. The declarations formalize the behavior of cryptographic primitives and their use is demonstrated on the OAUTH2.0 protocol . Process macros allow sub-processes to be defined, in order to ease development; and finally, the protocol itself can be encoded as a main process , with the use of macros.

**Declarations**

Processes are equipped with a finite set of types, free names, and constructors (function symbols) which are associated with a finite set of destructors. The language is strongly typed and user-defined types are declared as

**type t.**

All free names appearing within an input file must be declared using the syntax

**free n : t.**

where n is a name and t is its type. The syntax channel c. is a synonym for free c: channel. By default, free names are known by the adversary. Free names that are not known by the adversary must be declared private:

**Macros**

To facilitate development, protocols need not be encoded into a single main process . Instead, sub-processes may be specified in the declarations using macros of the form

**l e t R(x1 : t1, . . . , xn : tn) = P .**

where R is the macro name, P is the sub-process being defined, and x1, . . . , xn, of types t1, . . . , tn respectively, are the free variables of P. The macro expansion R(M1, . . . , Mn) will then expand to P with M1 substituted for x1, . . . , Mn substituted for xn. As an example, consider a variant docs/hello var.pv of docs/hello.pv

free c : ch annel .

free Cocks : b i t s t r i n g [ private ] .

free RSA: b i t s t r i n g [ private ] .

query a t t a c k e r ( Cocks ) .

l e t R( x : b i t s t r i n g ) = out( c , x ) ; 0 .

l e t R’ ( y : b i t s t r i n g )= 0 .

process R(RSA) | R’ ( Cocks )

By inspection of ProVerif’s output one can observe that this process is identical to the one in which the macro definitions are omitted and are instead expanded upon in the main process. It follows immediately that macros are only an encoding which we find particularly useful for development.

### 4.2.2WEBSPI LIBRARY AND ITS USAGE

Various calculi, starting from the spi-calculus , have been remarkably successful as modelling languages for cryptographic protocols, thanks also to the emergence of automated verification tools that can analyze large protocol models. Following in this tradition, we model web security mechanisms in an applied pi-calculus [3, 4], and verify them using ProVerif . We identify a set of idioms that are particularly useful in modeling web applications and web-based attackers, and o ffer them as a library, called WebSpi, available to other developers of web models.

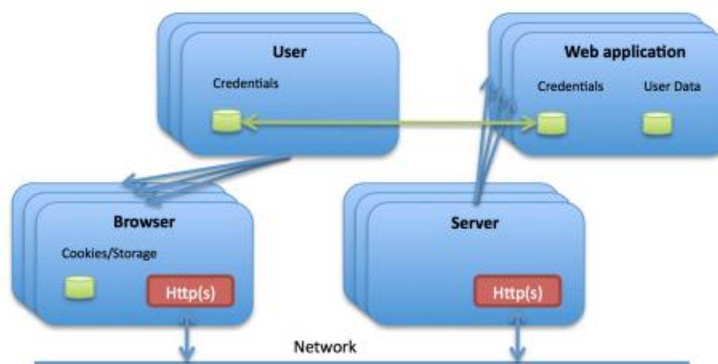



Figure 4.1 Webspi Architecture

### 4.2.3CRYPTOGRAPHY

ProVerif models symbolic cryptography: cryptographic algorithms are treated as perfect blackboxes whose properties are abstractly encoded using constructors (introduced by the fun keyword) and destructors (introduced by the reduc keyword). As an example, consider authenticated encryption:

**fun aenc(bitstring,symkey): bitstring**.

**reduc forall b:bitstring,k:symkey; adec(aenc(b,k),k) = b.**

Given a bit-string b and a symmetric key k, the term aenc(b,k) stands for the bitstring obtained by encrypting b under k. The destructor adec, given an authenticated encryption and the original symmetric key, evaluates to the original bit-string b.

ProVerif constructors are collision-free (one-one) functions and are only reversible if equipped with a corresponding destructor. Hence, MACs and hashes are modeled as irreversible constructors, and asymmetric cryptography is modeled using public and private keys:

**fun hash(bitstring): bitstring.**

**fun pk(privkey): pubkey.**

**fun wrap(symkey,pubkey): bitstring.**

**reduc forall k:symkey,dk:privkey; unwrap(wrap(k,pk(dk)),dk) = k.**

**fun sign(bitstring,privkey): bitstring.**

**reduc forall b:bitstring,sk:privkey; verify(sign(b,sk),pk(sk)) = b.**

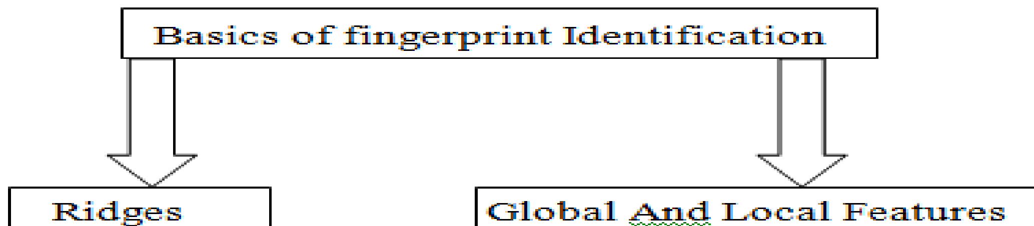
These and other standard cryptographic operations are part of the ProVerif library. Users can define other primitives when necessary. Such primitives can be used for example to build detailed models of protocols like TLS

## 4.3 DIGITAL PERSONA SOFTWARE

Digital Persona does not store copy of our fingerprint rather it it makes a map of 25-40 features and puts it in a data format called fingerprint template which can only be recognised by biometric engine.The template is a set of numbers difficult to recognise by the intruder.It uses 128 bit encryption key which is generated from the windows password which it prompts from every user at the time of login to encrypt the data into template which is regarded as tough task to crack.However ,considering the present scenario of parallel programming ,the computation power of computer can be enhanced to 90% that it has become possible to crack such encryption keys involving huge bits.Its all about computer algeabra.A team of researchers recently managed to crack 278 digits long 923 bit based cryptographic system.Moreover,the encryption key depends upon the strength of password set and normally users keep easy to remember passwords which is easier for intruder to get through.

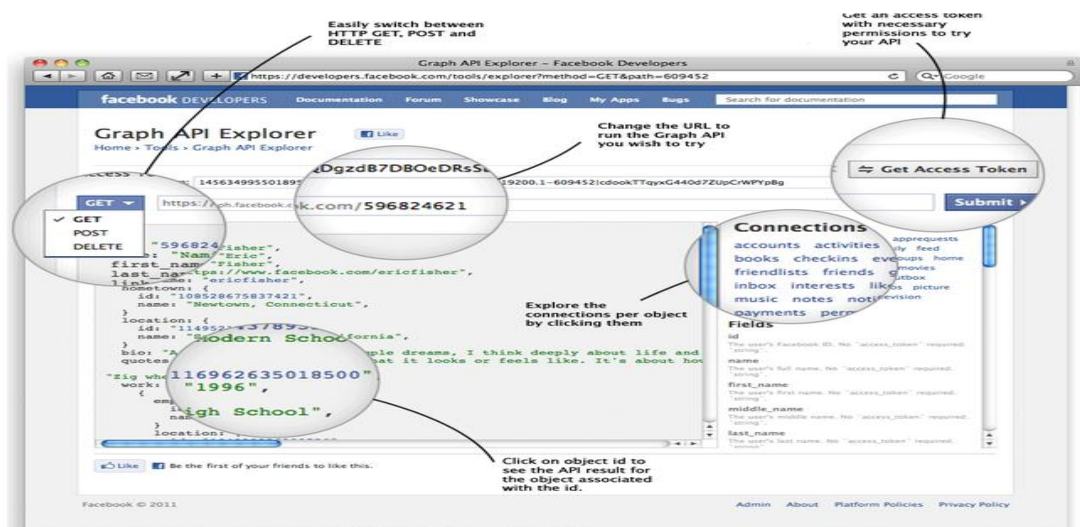
## 4.3.1WORKING

Of all the biometrics so used ,fingerprints is the most foolproof because an individual has 10 fingers each of which is unique




The Global features of an individual may be identical but local features such as minutae points which are tiny unique characteristics of fingerprint ridges are different for every individual[1,6].Hence,these are used for identification of user.The algorithm used is U.and.U fingerprint recognition algorithm.Its performance is measured as a tradeoff between two attributes-False Acception Rate(FAR) and False Rejection Rate (FRR).It is based on skeletal model and has been developed using machine learning techniques which makes it possible even to recognise the poor quality fingerprints as well.And even eliminates the latent fingerprints.Therefore ,a far and reliable software

## 4.4FACEBOOK GRAPH API EXPLORER



The Explorer helps you to do the following:

1. Make requests to the Graph API and see formatted results in-line.
2. Explore the connections for each object and view field descriptions to help you understand what the response means.
3. Easily obtain an access_token with the specific permissions necessary to access the data you need to optimize your app for users. The 'Select Permissions' dialog allows you to select the specific user data, friends' data or extended permissions needed. The Explorer then shows you the same permissions dialog your users will see when you request these permissions in your production app.

4.Easily switch between HTTP GET, POST and DELETE to get, create, update or delete objects. When POSTing to the API, you can provide a list of {name,value} pairs that you want to send as parameters to the API. For example to create a new event you can post to /me/events with the required name andstart_time fields as parameters, using an access_token with the create_event extended permission.

POST
https://graph.facebook.com/me/events
name
Test event
start_time
2011-06-24T10:00:00
Add a field

5.Move between objects in the graph just by clicking their id in the formatted result. For example clicking on the idof the event object returned after creating an event will show you details of the event you just created

## 4.5IMPLEMENTING OAUTH2.0 IN PROVERIF

### 4.5.1USER AGENT FLOW

The User-Agent flow, also called Implicit Grant flow, is meant to be used by client applications that can run JavaScript on the resource owner's user-agent. For example, it may be used by regular websites or by browser plugins. The authorization flow, adapted from the specification,is depicted .Let the resource server be located at the URL RS and its authorization server be located at AS. Let the resource owner RO have a username u and password p at AS. Let the client be located at URL C and have an application identifier id at AS. The message flow and relevant security events of the user-agent flow are as follows:

**1. Login(RO,b,sid,AS,u):** RO using browser b starts a login session sid at AS using credentials u,p.

**2. SocialLogin(RO,b,sid',C,AS,RS)**: RO using b starts a social sign-on session sid' at C using AS for RS.

**3. TokenRequest(C,b,AS,id,perms)**: C redirects b to AS requesting a token for id with access rights permsmissions.

**4. Authorize(RO,b,sid,C,perms):** AS looks up id and asks RO for authorization; RO using browser b in session sid at AS authorizes C with perms.

**5. TokenResponse(AS,b,C,token):** AS redirects b back to C with an access token.

**6. APIRequest(C,RS,token,getId()):** C makes an API request getId() to RS with token.

**7. APIResponse(RS,C,token,getId(),u):** RS verifies token, accepts the API request and returns u

to C.

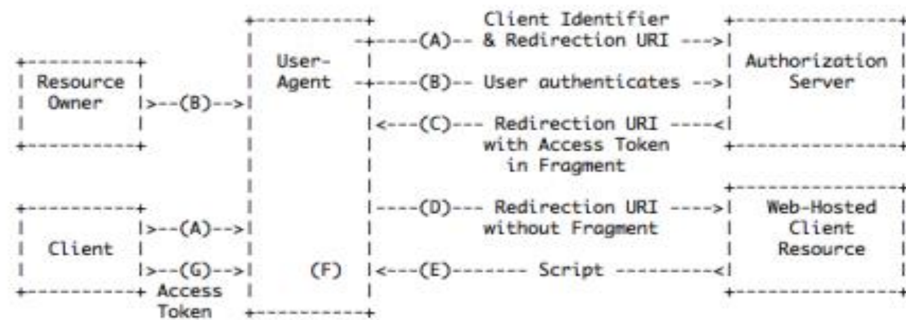
**8. SocialLoginAccept(C,sid',u,AS,RS):** C accepts RO's social sign-on session sid' as u at AS for RS.

**9. SocialLoginDone(RO,b,sid',C,u,AS,RS):** RO is logged in to C in a browser session sid' associated with u at AS, granting access to RS.

Here, Step 3 corresponds to action (A) in Figure 3; Step 4 corresponds to (B); Step 5 to (C)and (D); Step 6 onwards to (E), (F), and (G).

These steps may be followed by any number of API calls from the client to the resource server,on behalf of the resource owner. Each step in this flow consists of (at least) one HTTP requestresponse exchange. The specification requires that the AS must and the C should implement these exchanges over HTTPS. In the rest of this paper, we assume that all OAuth exchanges occur over HTTPS unless specified otherwise.As an example of the user-agent protocol flow, consider the social sign-on interaction between

websites like Pinterest and Facebook; the **TokenRequest(C,b,AS,id,perms)** step is typically implemented as an HTTPS redirect from Pinterest to a URI of the form:https://www.facebook.com/dialog/permissions.request?app_id=id&perms=email. The TokenResponse is also an HTTPS redirect back to Pinterest, of the form: https://pinterest.com/#access_token=token. Note that the access token is passed as a fragment URI. JavaScript running on behalf of the client can extract the token and then pass it to the client when necessary. In practice, these HTTP exchanges are implemented by a JavaScript SDK provided by Facebook and embedded into Pinterest, hence messages may have additional Facebook-specific parameters, but generally follow this pattern




## 4.5.2AUTHORIZATION CODE FLOW

The Authorization Code  flow (Figure 4), also called Explicit Grant flow or Web Server flow, can be used by client websites wishing to implement a deeper social integration with the resource server by using server-side API calls. It requires that the client must have a security association with the authorization server, using for example a shared secret. Moreover, it requires that the access token be retrieved on the server-side by the client. The motivation for this is two-fold:

(i) it allows the authorization server to authenticate the client's token request using a secret that only the client and the server know. In contrast, the authorization server in the user-agent flow has no way to ensure that the client in fact wanted a token to be issued, it simply sends a token to the client's HTTPS endpoint;

(ii) (ii) it prevents the access token from passing through the browser, and hence ensures that only the client application may access the resource server directly. In contrast, the access token in the user-agent
ow may be leaked in the Referer header, browser history, or by malicious third-party JavaScript running on the client.

The authorization flow is depicted in . Let the client at URL C have both an application identifier id and a secret sec pre-registered at AS. The diff erence between the web server and user-agent  flows begins after the SocialLogin step, and ends before the APIRequest step:

**3. CodeRequest(C,b,AS,id,perms):** C redirects b to AS requesting authorization for id with perms.

**4. Authorize(RO,b,sid,C,perms):** AS looks up id and asks RO for authorization; RO using browser b in session sid at AS authorizes C with perms.

**5. CodeResponse(AS,b,C,code):** AS redirects b back to C with an authorization code.

**5.1) APITokenRequest(C,AS,code,id,sec):** C makes an API request for an access token to AS with code, id, and sec.

**5.2)APITokenResponse(AS,C,token):** AS checks id and sec, verifies the code and returns a token to C

## 4.5.3 ADDITIONAL PROTOCOL PARAMETERS

In addition to the basic protocol flows outlined above, OAuth 2.0 enables several other optional features. Our models capture the following:

Redirection URI. Whenever a client sends a message to the authorization server, it may optionally provide a redirect uri parameter, where it wants the response to be sent. In particular, the TokenRequest and CodeRequest messages above may include this parameter, and if they do, then the corresponding APITokenRequest must also include it. The client may thus ask for the authorization server to redirect the browser to the same page (or state) from which the authorization request was issued. Since the security of OAuth crucially depends on the URI where codes and tokens are sent, the specification strongly advises that clients must register all their potential redirection URIs beforehand at the authorization server. If not, it predicts attacks where a malicious website may be able to acquire codes or tokens and break the security of the protocol. Indeed, our analysis found such attacks both in our model and in real websites. We call such attacks Token Redirection attacks.

State Parameter. After the TokenRequest or CodeRequest steps above, the client waits for the authorization server to send a response. The client has no way of authenticating this response, so a malicious website can fool the resource owner into

sending the client a different authorization code or access token (belonging to a di fferent user). This is a variation of the standard website login CSRF attack that we call a Social Login CSRF attack. To prevent this attack, the OAuth specification recommends that clients generate a nonce that is strongly bound to the resource owner's session at the client (say, by hashing a cookie). It should then pass this nonce as an additional state parameter in the CodeRequest or TokenRequest messages. The authorization server simply returns this parameter in its response, and by checking that the two match, the client can verify that the returned token or code is meant for the current session. After incorporating the above parameters, the following protocol steps are modified as shown:

**TokenRequest(C,b,AS,id,perms,state,redirect uri)**
**TokenResponse(AS,b,redirect uri,state,token)**
**CodeRequest(C,b,AS,id,perms,state,redirect uri)**
**CodeResponse(AS,b,redirect uri,state,code)**
**APITokenRequest(C,AS,code,id,sec,redirect uri)**
**APITokenResponse(AS,C,token)**

Our analysis does not cover other features of OAuth, such as refresh tokens, token and code expiry, the right use of permissions, or the other protocol flows described in the specification.

We leave these features for future work

## 4.5.4THREAT MODEL

The OAuth specification describing its threat model  together provide an exhaustive list of potential threats to the protocol. We consider a subset of these threats in our formal analysis. The ultimate aim of the attackers we consider is to steal or modify the private information of an honest resource owner, for example by fooling honest or buggy clients, authorization servers,or resource owners into divulging this information. To this end, we consider: network based attackers who can sniff, intercept, and inject messages into insecure HTTP traffic; malicious clients, resource owners, and authorization servers; malicious websites that honest resource owners may browse to; and honest clients with specific web vulnerabilities, such as CSRF

attacks, or redirectors that may forward HTTP requests to malicious websites. We do not explicitly consider attacks on the browser or operating system of honest participants; instead, we treat such participants as compromised, that is, as fully controlled by an attacker. We assume that honest resource owners choose strong passwords and use secure web browsers. We assume that honest authorization servers have no web vulnerabilities, instead we focus on vulnerabilities in client websites.

## 4.6SECURITY GOALS

We describe the security goals for each participant by defining Datalog-like authorization policies that must be satisfied at di fferent stages of the protocol. The policy A : B,C is read as A if B and C".The resource owner RO (using browser b) in a session sid' with a client C has successfullycompleted the social sign-on with authorization server AS (and resource server RS) if it intended to sign into the client, if it agreed to authorize the client, and if the client and resource owner agree upon the user's social identity (u) for the current session (sid'):

**SocialLoginDone(RO,b,sid',C,u,AS,RS) :**
**Login(RO,b,sid,AS,u),**
**SocialLogin(RO,b,sid',C,AS,RS),**
**Authorize(RO,b,sid,C,idPermission),**
**Says(C,SocialLoginAccept(C,sid',u,AS,RS)).**

The authorization server must ensure that a token is issued only to authorized clients. Its policy for the user-agent flow says that a TokenResponse can only be sent to C if the resource owner has logged in and authorized the client.

**TokenResponse(AS,b,C,state,token) :**

**ValidToken(token,AS,u,perms),**
**Says(RO,Login(RO,b,sid,AS,u)),**
**ValidClient(C,id,redirect uri),**
**Says(RO,Authorize(RO,b,sid,C,perms)).**

Note that we do not require a TokenResponse to be only issued in response to a TokenRequest from the client: at this stage, the user-agent flow has not authenticated the client, and so cannot know whether the client intended to request a token.The correponding policy for the authorization code flow is stronger:

**APITokenResponse(AS,C,state,token) :**
**ValidToken(token,AS,u,perms),**
**Says(RO,Login(RO,b,sid,AS,u)),**
**ValidClient(C,id,redirect uri),**
**Says(C,TokenRequest(C,b,AS,id,perms,state,redirect uri)),**
**Says(RO,Authorize(RO,b,sid,C,perms)).**

From the viewpoint of the resource server, every API call must be issued by an authorized client and accompanied by a token issued by the authorization server.

**APIResponse(RS,b,C,token,req,resp) :**
**ValidToken(token,AS,u,perms),**
**Permitted(perms,req),**
**Says(C,APIRequest(C,RS,token,req)).**

Finally, from the viewpoint of the client, the social sign-on has completed successfully if it has correctly identified the resource owner currently visiting its page, and obtained an access token for the API accesses it requires.

**SocialLoginAccept(C,sid',u,AS,RS) :**
**Says(RO,SocialLogin(RO,b,sid',C,AS,RS)),**
**Says(AS,TokenResponse(AS,b,C,token)),**
**Says(RS,APIResponse(RS,C,token,getId(),u))**

### 4.7MODELLING PRINCIPALS

The agents are called principals. They can play the role of users or owners of web applications.Hence, the same principal may own two different web applications and be the user of a third one. Users hold credentials to authenticate with respect to a specific web application (identified by a host domain and subdomain) in the table

credentials. Web applications hold  private and public keys to implement TLS secure **table credentials(Host,Principal,Id,Credentials)**
**table serverIdentities(Host,Principal,pubkey,privkey,XdrFlag).onnections in the table serverIdentities**.

| **PRINCIPALS** | |
|---|---|
| HTTP REQUEST | httpReq(uri(protocol,domainHost(subdomain,domain),path,params), headers(referrer,cookies,notajax()),httpGet()) |
| HTTP SERVER | let HttpServer() =in(net,(b:Browser,o:Origin,m:bitstring));get serverIdentities(=originhost(o),pr,pk P,sk P,xdrp) in<br><br>let (k:symkey,httpReq(u,hs,req)) = reqdec(o,m,sk P) in<br><br>if origin(u) = o then<br><br>let corr = mkCorrelator(k) in<br><br>out(httpServerRequest,(u,hs,req,corr));<br><br>in(httpServerResponse,(=u,resp:HttpResponse,cookieOut:CookieSet ,=corr));<br><br>out(net,(o,b,respenc(o,httpResp(resp,cookieOut,xdrp),k))) |
| HTTP RESPONSE | (let httpOk(dataIn) = resp in<br><br>if p = aboutBlank() then<br><br>(let p1 = mkPage(u) in<br><br>insert pageOrigin(p1,o,h,u);<br><br>out (newPage(b),(p1,u,dataIn)))<br><br>else<br><br>(if aj = ajax() then<br><br>(get pageOrigin(=p,oldorig,oldh,olduri) in |

|  |  |
|---|---|
|  | if (foo = xdr() jj oldorig = o) then<br><br>out (ajaxResponse(b),(p,u,dataIn)))))<br><br>j(let httpRedirect(redir) = resp in<br><br>out (internalRequest(b),(redir,httpGet(),ref,p,notajax()))))) |
| COOKIE RESET | (get pageOrigin(p,o,h,ref) in<br><br>get cookies(=b,=originhost(o),=slash(),cs) in<br><br>get cookies(=b,=originhost(o),=h,ch) in<br><br>get storage(=b,=o,s) in<br><br>out<br>(getCookieStorage(b),(p,cookiePair(protocolCookie(domcookie(cs),o),<br><br>protocolCookie(domcookie(ch),o)),s)))<br><br>Cookies are indexed by origin and by path (where slash() stands for the empty path). |
|  |  |

|  |  |
|---|---|
| LOGIN USERPROCESS | let LoginUserAgent(b:Browser) =<br><br>let p = principal(b) in<br><br>in(newPage(b),(p1:Page, u:Uri, d:bitstring));<br><br>let (=https(), h:Host, loginPath(app)) = (protocol(u),host(u),path(u)) in<br><br>(( |

| | |
|---|---|
| | let loginForm(as) = formTag(d) in<br><br>get credentials(=h,=p,uId,pwd) in<br><br>if assume(Login(p,b,h,uId)) then<br><br>out(pageClick(b),(p1,u,httpPost(loginFormReply(d,uId,pwd))))<br><br>)j(<br><br>if loginSuccess() = d then<br><br>event Expect(ValidSession(p,b,h))) |
| LOGIN<br>SERVER<br>PROCESS | let LoginApp(h:Host,app:Path) =<br><br>in(httpServerRequest,(u:Uri,hs:Headers,req:HttpRequest,corr:bitstring));<br><br>let uri(=https(),=h,=loginPath(app),q) = u in<br><br>et c = getCookie(hs) in<br><br>let cookiePair(sid,ch) = c in<br><br>let httpPost(loginFormReply(d,uId,pwd)) = req in<br><br>get credentials(=h,p,=uId,=pwd) in<br><br>get serverIdentities(=h,sp,xx,yy,zz) in<br><br>event Expect(LoginAuthorized(sp,h,uId,sid));<br><br>insert serverSessions(h,sid,loggedIn(uId));<br><br>out(httpServerResponse,(u,httpOk(loginSuccess()),c,corr)) |
| LOGIN<br>AUTHORISED | LoginAuthorized(sp,h,uId,sid) :<br><br>Server(sp,h), |

| | |
|---|---|
| | User(up,uId,h),<br><br>Says(up,Login(up,b,h,uId)) |
| VALID SESSION | ValidSession(up,b,h) :<br><br>Server(sp,h),<br><br>User(up,uId,h),<br><br>Login(up,b,h,uId),<br><br>Says(sp,LoginAuthorized(sp,h,uId,sid)) |

4.8ATTACKER MODEL

| | | |
|---|---|---|
| *Principals* | createServer(sp) | create a new server for principal $sp$ |
| | createUser(up,h,p) | create a new user $up$ for the app at path $p$ on host $h$ |
| | compromiseUser(id,h,p) | force user with login $id$ on app $p$ at $h$ to reveal its password |
| | compromiseServer(h) | force principal of server hosted at $h$ to reveal its secret key |
| *Network* | injectMessage(e1,e2,m) | send message $m$ to endpoint $e2$ as if it came from $e1$ |
| | interceptMessage(e1,e2) | intercept a message from $e1$ to $e2$ |
| *Websites* | startUntrustedApp(h,p) | start a malicious application $p$ at $h$ |
| | getServerRequest(h,p) | intercept a request between the http module and app $p$ at $h$ |
| | sendServerResponse(h,p,u,r,c,m) | send $m$ to $u$ on behalf of $h, p$, with cookie $c$ and HTTP response type $r$, from the server with principal $sp$ |
| | httpRequestResponse(c,u,m) | send $m$ to $u$ and wait for response |
| *JavaScript* | getClientResponse(b,h,p) | intercept the response from browser $b$ to app $h, p$ |
| | sendClientRequest(b,h,p,c,u1,u2,m) | send $m$ to $h, p$ as if $b$ clicked on $u1$ on a page from $u2$ |

We consider a standard symbolic active (Dolev-Yao) attacker who controls all public channels and some principals, but cannot guess secrets or access private channels. Furthermore, the attacker can create new data and can encrypt or decrypt any message for which it has obtained the cryptographic key, but otherwise cannot break cryptography.

By default, all the channels, tables, and credentials used in WebSpi are private. We define a process AttackerProxy that mediates the attacker's access to these resources, based on a set of configuration flags. The attacker executes a command by sending a message on the public channel admin and if the current configuration allows it, the process executes the command and returns the result (if any) on the public channel result:

**let AttackerProxy() =**

```
in (pub,x:Command);
if commandEnabled(x) = true then
out(admin,x);
in (result,(=x,y:bitstring));
out(pub,y).
```

The full list of commands that the attacker can send is listed in Table 1. This API is designed to be operational: each command corresponds to a concrete attack that can be mounted on a real web interaction. It includes three categories of attacker capabilities

**f**

**nes**

**several di**

**erent**

**ows or protocol con**

**gurations, two of which directly apply to website applications. The protocol itself requires no cryptographic mechanisms whatsoever and instead relies on transport layer security (HTTPS). Hence, it claims to be lightweight and exible, and has**

## 3.1 TERMINOLOGY

**client**

An HTTP client (per [RFC2616]) capable of making OAuth- authenticated requests

**Server**

An HTTP server (per [RFC2616]) capable of accepting OAuth- authenticated requests protected resource An access-restricted resource that can be obtained from

the server using an OAuth-authenticated request (Section 3).

**Resource owner**

An entity capable of accessing and controlling protected resources by using credentials to authenticate with the server.

**Credentials**

Credentials are a pair of a unique identifier and a matching shared secret. OAuth defines three classes of credentials: client, temporary, and token, used to identify and authenticate the client making the request, the authorization request, and the access grant, respectively.

client

An HTTP client (per [RFC2616]) capable of making OAuth-

authenticated requests (Section 3). Server An HTTP server (per [RFC2616]) capable of accepting OAuth-authenticated requests (Section 3). protected resource An access-restricted resource that can be obtained from the server using an OAuth-authenticated request (Section 3). resource owner An entity capable of accessing and controlling protected resources by using credentials to authenticate with the server. Credentials are a pair of a unique identifier and a matching shared secret. OAuth defines three classes of credentials: the client making the request, the authorization request, and the access grant, respectively

Credentials

X

.