

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/276397514>

Security evaluation of the OAuth 2.0 framework

Article in *Information and Computer Security* · March 2015

DOI: 10.1108/ICS-12-2013-0089

CITATIONS

23

READS

10,957

3 authors, including:



John O'Raw

Letterkenny Institute of Technology

19 PUBLICATIONS 154 CITATIONS

[SEE PROFILE](#)



Kevin Curran

Ulster University

577 PUBLICATIONS 7,670 CITATIONS

[SEE PROFILE](#)

Security Evaluation of the OAuth 2.0 Framework

Kevin Gibbons, John O Raw

School of Computing, Letterkenny Institute of Technology, Donegal, Ireland

Kevin Curran

School of Computing and Intelligent Systems, University of Ulster, Northern Ireland

Email: kevin.curran@ulster.ac.uk

Abstract

The interoperability of cloud data between web applications and mobile devices has vastly improved over recent years. The popularity of social media, smartphones and cloud based web services have contributed to the level of integration that can be achieved between applications. This paper investigates the potential security issues of OAuth, an authorisation framework for granting third party applications revocable access to user data. OAuth has rapidly become an interim defacto standard for protecting access to web API data. Vendors have implemented OAuth before the open standard was officially published. To evaluate whether the OAuth 2.0 specification is truly ready for industry application, an entire OAuth client server environment was developed and validated against the specification threat model. The research also included the analysis of the security features of several popular OAuth integrated websites and comparing those to the threat model. High impacting exploits leading to account hijacking were identified with a number of major online publications. It is hypothesised that the OAuth 2.0 specification can be a secure authorisation mechanism when implemented correctly.

1. Introduction

Integration between applications has become a commodity in recent years due to the rising popularity of cloud API services, social media and mobile computing. This innovation has also brought new security threats. It is difficult to truly verify that someone is who they claim to be. Authentication is an issue dating thousands of years and today, authentication is still a challenging problem. Authorising an application access to user data from another application is no exception. Authentication is the process of validating that a user is who they claim to be. In contrast, authorisation limits the data or actions an authenticated user can access. Authentication has always been an issue in computing. As technology has advanced so too have authentication methods and ways to exploit them. There have been several notable cases of compromised user accounts in recent history. Most notably, the attack launched against journalist Mat Honan where his Google, Twitter and Apple accounts were all compromised as well as his MacBook [1]. Attackers targeted the journalist to get access to his Twitter account. After gaining access to his Amazon account (method unknown) the attackers were able to view the last four digits of his credit card number. The attackers were then able to reset Mat's Apple account password via Apple technical support using the same last four digits as verification. From the Apple account the attackers could then access his Gmail account and from there his Twitter account. The attackers were able to wipe contents of the journalist's iPhone, iPad and MacBook as his Apple account supported iCloud, Apple's cloud storage service. Though this attack, like most, was primarily due to social engineering and lack of universal security policies between organisations, it shows that although security advancements have been made organisations are still not implementing authentication properly.

Authentication methods used today include textual passwords, graphical passwords, 3D passwords, third party federation and biometrics [2]. Each method has its advantages and disadvantages and compromise between

security and usability. Google's opt in two step verification greatly mitigates the likelihood of an account being compromised. Multi factor authentication adds another layer of security by sending a verification code to the user's mobile phone or other device when logging in from an unknown machine which the user must then input back to Google. Mat Honan stated that if he had implemented Google's two step verification then the whole attack may have been averted [1]. Google reported that almost 250,000 users added two step verification to their account within two days of Mat Honan publishing his article [3]. The chances of an attacker stealing both a user's login credentials and mobile phone are relatively small. One disadvantage of this opt in protection is that attackers who have gained access to Google accounts can enable the service with their own mobile number to impede the user restoring their account. A new method of authenticating access to cloud data APIs has emerged called OAuth. This authorisation framework grants third parties access to user data without the user disclosing their credentials to the third party service [4]. OAuth has already been widely adopted by industry players and is on track to becoming a standard for authorisation. OAuth seeks to resolve the shortcomings of propriety authentication protocols by creating a universal and interoperable authorisation mechanism between services. OAuth can also be used to protect user information in Single Sign On (SSO) mechanisms. SSO lets the user register or login temporarily with a new website using the profile information already stored on another service. Major online brands, such as Facebook and Google federate identification in this manner. Single Sign On provides many benefits for user. They have fewer passwords to remember. It also stops the use of the same password for multiple accounts. The user also does not have to complete tedious and repetitive registration forms. An overall richer user experience is gained with SSO. Disadvantages include a new platform for phishing attacks and a single point of failure. If the identity provider account was compromised an attacker would have access to all the victim's linked accounts.

OAuth has rapidly become an interim authentication and identity management standard for online applications. As social network trends soared, many sites were quick to adopt OAuth but did not implement the framework to the specification and security aspects were overlooked. This paper investigates the security of the OAuth authorisation framework and compares it to current industry implementations to evaluate if OAuth is truly ready for widespread adoption. Severe security vulnerabilities were identified, one of which resulted in user account hijacking. An OAuth client and server were developed to test the robustness of the specification.

2. OAuth

As data has migrated to the cloud vast interoperability options have been realised by online services. Access to that data has also become ubiquitous via smartphone and tablet computing. A universal mechanism was required so that these services and devices could access this data without compromising security, incurring development complexity and provide a consistent user experience. This requirement came in the form of OAuth, an authorisation framework that is soon becoming a de facto standard for online authentication. OAuth is the authentication method of choice by social sites Facebook and Twitter to protect their APIs and federate identification across domains [5]. OAuth moves away from the vulnerable username-password paradigm and has adopted the use of a bearer token. The Internet Engineering Task Force (IETF) administers the open standard and have published the 2.0 version of the specification [4]. OAuth allows a third party service to access user data on another service without disclosing the login credentials for that service. For example, a user can authorise HP's online photo printing service SnapFish to access their images on their Facebook account without giving SnapFish their Facebook password.

The analogy of a car valet key is often used to describe OAuth. A valet key restricts the temporary driver to a certain speed limit and maximum distance, the glove box and boot may also be inaccessible with the valet key. Similarly, OAuth can enforce restrictions such as access scope and token expiration to limit clients only to certain data or functions for a specified time [6]. The prevalence of cloud APIs and online services in recent years and the need to integrate them has demanded a common protocol for delegating authorisation. Demand has also spread to mobile computing, in the form of smart phones and tablets, which are soon becoming personal commodities. In 2012, a total of 700 million mobile devices were shipped worldwide [7]. The increase in social media usage has also greatly contributed to the popularity of OAuth. Social network giants such as Facebook and Twitter use OAuth to authenticate their REST APIs for integration with other applications. In 2012, 67% of American adult internet users used social networking accounts [8]. These social sites are almost synonymous with the Internet, so much so that they are now becoming federators of identification.

2.1 The Demand for OAuth

OAuth was developed to create a common authorisation mechanism that could be used across many online services [9, 10, 11, 12]. Before OAuth, developers had to build custom implementations for each service's proprietary authorisation method, such as Google ClientLogin and Facebook FBAuth [13]. Supplying a username and password to a third party service would have many detrimental effects. The user may not be comfortable disclosing their password to another service and it goes against the efforts that have been made in phishing awareness. Unfortunately many users still use a common password across multiple online accounts. If the service was ever compromised the attacker could access all the user's accounts. 58% of users reuse the same password [14]. Providing a password also gives the third party service full access to the user account which could be used maliciously. This method also does not support any revocation mechanism apart from changing the password which would also revoke any other third party services that also accessed that account. A user could also change their password without realising that they will revoke access from their third party services. SAML and OpenID do not use passwords and therefore would not be compatible with the service. OAuth alleviates all these issues through the notion of a temporary access token. An access token is supplied to the resource provider instead of a username and password. The access token is associated with a client, the scope that the client is limited to and an expiry date.

The OAuth 2.0 specification has defined several profiles for varying scenarios [15]. These profiles include a server side flow for authenticating access to server side applications, a client side flow for JavaScript or other client based applications and a native flow for desktop or mobile applications. The OAuth 2.0 specification has been extended to allow for the integration of additional authentication mechanisms. OAuth can be built on top of pre-existing authentication mechanisms such as an LDAP directory or SAML [16].

2.2 OAuth and Identity Providers

There is a misconception that OAuth can federate identification on its own. OAuth is merely the authorisation mechanism to an identity provider's resource server [17]. OAuth coupled with OpenID allows sites like Facebook and Twitter to provide identification. OpenID is another open standard which provides identify information to authenticated clients. OpenID allows users whom are already registered to a trusted site, such as Facebook, to sign up to other websites without having to create another set of credentials. Instead of a user entering a name, email, password, username and so on they can simply "Sign In with Facebook". The latest specification is called OpenID Connect which utilises OAuth for authentication [17]. An OpenID Connect authorisation follows the usual OAuth sequence. A client application (relying party) registers with an identity provider service, such as Google. When a user wishes to sign into the client application for the first time, they will be redirected to Google for login and consent. Upon user consent Google will redirect back to the client application with an authorisation code, which is exchanged for an access tokens. As user profile information would have been requested in the scope parameter, an id token will also be returned with the access token. The id token represents a base64url encoded JWT security token. The JWT token has 3 parts, header, claim set data and digital signature, delimited by the '.' character.

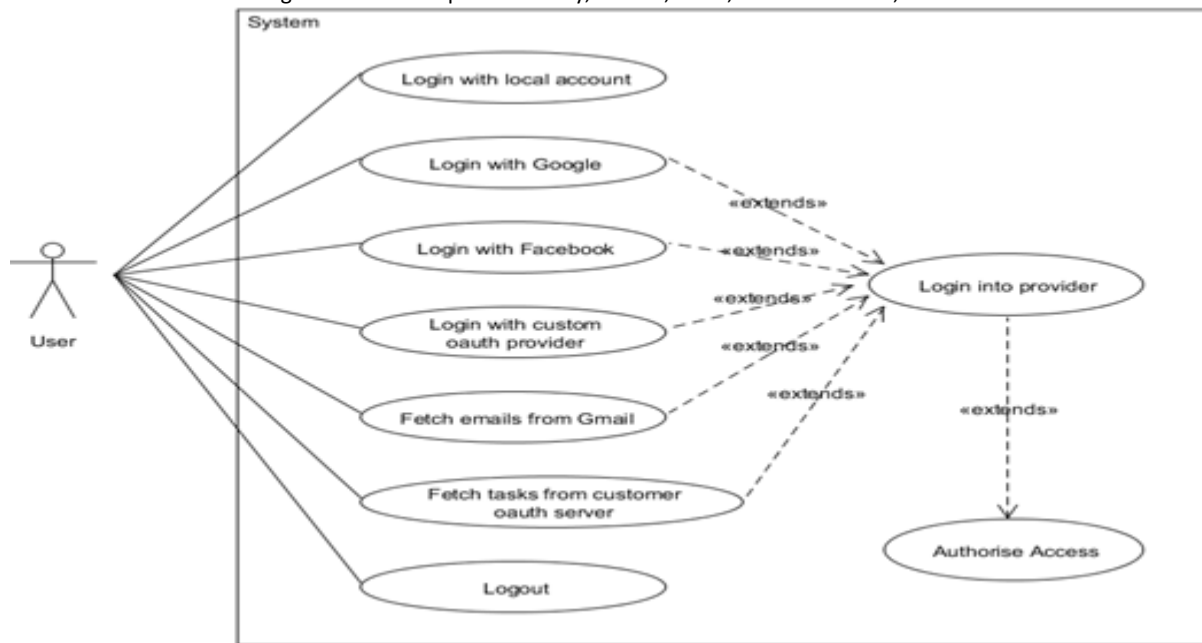


Figure 1: OAuth Server Side Process

Per the specification, the client application must validate the received id token with the authorisation server's token info endpoint (see Figure 1). This is to mitigate replay attacks. The whole token, as it was received, is sent to the token info endpoint and a JSON object is returned containing information about the client the token was issued for. The client then has to decode the JWT and compare the values to the values returned from the token info point. The client has to ensure that the audience values match. The client must also validate timestamps for equality. Upon successful validation, the client can query the identity provider's user info endpoint with the access token to retrieve further information about the user, such as email address, gender, date of birth and profile picture. This single sign on mechanism alleviates populating redundant registration forms and provides a richer user experience. As SSO is becoming more prevalent, relying party applications are becoming more customizable based on the way the user has authenticated and the account they used. [18] has identified usage patterns and the challenges in mapping provider account data to the custom user attributes in the client application. The IETF working group have outlined several potential security issues with the standard and how they should be mitigated [19]. The threat model is broken down into sections based on the different locations in an OAuth environment, parameters, and extensions. Most of the issues are resolved through the use of HTTPS and validating the client at every request.

2.3 Future of OAuth

The IETF are currently working on a draft specification for dynamic client registration [20] for allowing clients who were not previously registered with an authorisation server to register on the fly before user authentication. The mechanism involves the authorisation server providing client registration and configuration endpoints. In the proposed solution, the client would make a request to the authorisation server's client registration endpoint supplying the necessary criteria. The authorisation server should then return a client id generated for client and optionally a client secret depending on the registration data provided. A registration access token will also be returned which the client application can use to modify information about itself at the client configuration endpoint URI on the authorisation server which is also returned in the payload as well as timestamp data. The draft is scheduled for standardisation submission for Autumn 2014 [21]. OpenID Connect also have a draft specification for dynamic client registration based on IETF's draft [22]. Identity provider Cloud Identity supports a dynamic client registration implementation [23].

OAuth 2.0 also has a draft specification for using Message Authentication Code (MAC) tokens instead of a bearer token [2424]. MAC tokens add another layer of security by binding the access token to a client and resource server via a symmetric session key. MAC tokens are protected from replay attacks and accidental leakage as they

can only be used for the intended client and resource server. Implementing MAC tokens augments the complexity of the OAuth process and therefore has not been greatly adopted in industry though the specification is still in draft changes and is subject to change. Several papers have investigated potential enhancements to OAuth performance and usability. In [25], two optimisation steps were proposed to reduce the number of requests in the authorisation sequence. The first involves caching authorisation data in a table on the authorisation server, resource server and client. The client table contains the URL to the authorisation endpoint so that it can access it directly instead being redirected when making a request to the resource server. The second optimisation introduced an "AppliesTo" parameter, so that access tokens that were applicable to all users could be saved by the client and reused when other users requested access instead going to the authorisation server again with a grant request. Though the reduced number of HTTP requests and mitigated DoS attacks sound promising the paper fails to examine the potential security issues with their proposed improvements. Having more data spread across multiple sites increases the attack entry points to sensitive information. Stored URLs goes against the flexibility of the specification. Though the "AppliesTo" parameter could potentially save a lot of HTTP requests this is outweighed by the huge security vulnerability. If an attacker somehow gained access to one employee's access token he would then be able to act on behalf of all the employees in the organization.

Shehap and Marouf [26] proposed an extension to the OAuth framework which provided finer grained scope and access recommendations based on previous authorisations. A browser extension and algorithm was developed to suggest which permissions the user should select when granting access to their data. They concluded that users were less likely to select all permissions when recommendations were suggested. They also acknowledged that the threshold value could be inaccurate if the same application was repeatedly used or if the underlying system was compromised. Google's VP of Security Engineering, Eric Grosse, envisioned a potential improvement of the OAuth specification involving channel binding [3]. Channel binding eliminates man in the middle attacks and session hijacking by cryptographically binding cookies to a Transport Layer Security (TLS) channel ID [27]. The Google Chrome browser implements channel binding [27]. Research is currently being carried out to make authentication more interactive with the user. [28] has investigated the idea of integrating biometrics with OAuth so that the user can grant consent based on the prints of ones knuckles. High accuracy was achieved and only standard hardware, such as a web camera, was required. Google are also looking towards making authentication more personal. They are experimenting with USB tokens, a USB device in the user's machine which acts like an ignition key to protected resources [3]. The token communicates signed assertions to compliant browsers and provides new layers of security. To alleviate the burden of the user having to remember to carry around a USB device Google are looking at ways to combine USB tokens with ubiquitous hardware such as a smartphones or "smart jewellery". They are currently researching the use of wireless technology, Bluetooth and NFC, so that users can authenticate applications and other devices with their smartphone or a tap from their "smart ring" [3].

The OAuth 2.0 specification is still very much in its infancy, but when implemented correctly it can provide a relatively secure and interoperable authentication delegation mechanism. The IETF are currently addressing issues and expansions in their working drafts. Once a strict level of conformity is achieved between vendors and vulnerabilities are mitigated it is likely the framework will change the way we access data on the web and other devices.

3. Exploiting OAuth

Most of the known OAuth 2.0 exploits have been due to the implementation, not the framework itself. The IETF have predicted many of the security issues in the threat model but these have been overlooked by several sites during the recent popularity of OAuth and social media integration. OAuth 2.0 is vulnerable to CSRF (Cross Site Request Forgery) attacks when the state parameter is not implemented [13], [28]. A CSRF attack is where a user unknowingly makes a request (clicking a link or downloading an image) of malicious nature to a web site that the browser is currently logged in to. For example, a user may be logged into their online banking in one tab and open a link in an email. The malicious link navigates to the online banking's funds transfer endpoint with the appropriate URL parameters to wire the user's money to another account. As the user is logged into the online banking in one tab, the browser cookie has been set and so it will be sent in the request header of the malicious request. To the online banking server, the request appears as if it genuinely came from the user.

CSRF attacks can be applied to OAuth in the following manner. Consider a user who is currently logged into a service like Pinterest which supports SSO with Facebook or Twitter. An attacker would sign into Pinterest with Facebook to obtain a redirect URL with a valid authorisation code. The attacker will not navigate to this redirect link however as this would nullify the authorisation code. The attacker could then send an email to the victim containing a link to a malicious web page. The malicious web page could contain an image with the source attribute pointing to Pinterest's OAuth redirect URL with the valid authorisation code associated with the attacker's Facebook account. If the user was logged into Pinterest while opening the email link the browser would make the "image" request with the corresponding cookie for the session. The user will have unknowingly granted the attacker access to their Pinterest account. The attacker would then be able to sign into Pinterest with his Facebook credentials and access the user's data. This issue is identified in 4.4.1.8 of the threat model [19] which recommends utilising the state parameter. This mitigates the threat as the callback URL must contain the same state parameter value that was sent in the authorisation request. State parameters are high entropy nonces so they would be difficult for an attacker to emulate. In the early days of OAuth 2.0 many sites like Pinterest and TripAdvisor were vulnerable to this attack though they have now mitigated the issue via CSRF tokens and the use of the 'state' parameter.

A formal analysis model for OAuth 2.0 was devised in [2830] utilising the automated security protocol evaluation tool ProVerif. Several vulnerabilities with popular vendors, such as Yahoo and WordPress, were identified. An exploit was found with Yahoo's single sign on implementation. If a user who was logged into Yahoo via Facebook navigated to a malicious site it could redirect the user to Facebook's authorisation endpoint with Yahoo's client ID and a redirect URI. As the user is already logged in authorisation has already been granted and so Facebook redirects the user to the provided redirect URI with a valid access token. The attacker could exploit Yahoo's open redirector mechanism by appending a redirect URL parameter to the redirect URL, e.g. `http://yahoo.com/r/_ylt=A7x9.../**http://evil.com`. Facebook would return to the Yahoo redirect URL which would then redirect to the malicious site with the access token included. The malicious site could then use the token on the resource server to act on behalf as the user.

Several vulnerabilities were identified in cloud storage including another open redirector exploit which could redirect Facebook access tokens [29]. OAuth can also be exploited by clickjacking, the action of tricking the user into clicking on a UI element that's been masqueraded as another [30]. A malicious site could use CSS to overlay and hide an OAuth consent dialog in a frame on top another UI. The non-visible "OK" button in the consent dialog would be placed directly above another button, such as "Win a free iPad" or similar. The user of course would have to be logged into the service in order to render the hidden dialog. When the user clicks the "Win a free iPad" button they inadvertently authorise the malicious site access to their account. Though this exploit is not directly the fault of the OAuth process. Clickjacking is mentioned in part 4.4.1.9 of the threat model [19]. The use of the X-FRAME-OPTIONS HTTP header is the recommended mitigation. This header allows servers to specify whether the content can be displayed in a frame. Facebook's OAuth consent dialog uses the x-FRAME-OPTIONS value "deny" so that it cannot be placed in a frame. It should be noted that this HTTP header field is not supported in older browsers. OAuth supports offline access via refresh tokens though it is not mandatory [4]. Offline access is convenient for applications and has legitimate purpose but it can also be exploited. Social bots which implement offline access can be used to log in and imitate human social network accounts and automate "friend requests" as a means of data harvesting [31]. When a user accepts a "friend request" or similar connection, their personal information, such as email address and date of birth, are available to the social bot. Social bots are also used for mass distribution of spam and propaganda. Again, this issue is not the fault of the framework as it is being utilised correctly, though for ill purpose. The OAuth 2.0 specification states that supporting refresh tokens is optional. They have also researched detecting botnet behaviour in social networks [32].

3.1 Real World OAuth Exploit Tests

To analyse the security of OAuth implementations in industry a list of the fifty most popular websites in Ireland was retrieved from the statistical website Alexa [35]. Each site was analysed to identify if it utilised OAuth. Out of the fifty sites, 21 were identified with OAuth support. Each vulnerability in the threat model was then tested against each OAuth enabled site. Two sites out of the 21 were found to be susceptible to some form of attack meaning that 10.5% were vulnerable. 18% of the world's 50 most popular sites were in the list of 21 OAuth enabled sites.

	Name	OAuth Enabled	Exploit Found		Name	OAuth Enabled	Exploit Found
1	Google.ie	✓	✗	26	Dailymail.co.uk	✓	✓
2	Google.com	✓	✗	27	Google.co.uk	✓	✗
3	Facebook.com	✓	✗	28	Adverts.ie	✓	✗
4	Youtube.com	✓	✗	29	eBay.co.uk	✗	✗
5	Yahoo.com	✓	✗	30	365online.com	✗	✗
6	Wikipedia.com	✗	✗	31	Googleusercontent.com	✗	✗
7	LinkedIn.com	✓	✗	32	msn.com	✗	✗
8	Live.com	✓	✗	33	<omitted>	✗	✗
9	Twitter.com	✓	✗	34	Avg.com	✗	✗
10	Amazon.co.uk	✓	✗	35	Thepiratebay.se	✗	✗
11	Amazon.com	✓	✗	36	Vodafone.ie	✗	✗
12	Aib.ie	✗	✗	37	Microsoft.com	✗	✗
13	Rte.ie	✗	✗	38	Bing.com	✗	✗
14	Bbc.co.uk	✗	✗	39	Flickr.com*	✓	✗
15	DoneDeal.ie	✗	✗	40	Gaurdian.co.uk	✓	✓
16	Boards.ie	✗	✗	41	O2online.ie	✗	✗
17	eBay.ie	✗	✗	42	Imgur.com	✓	✗
18	WordPress.com	✓	✗	43	Apple.com	✗	✗
19	Independent.ie	✓	✗	44	eBay.com	✗	✗
20	PayPal.com	✗	✗	45	Travian.co.uk	✗	✗
21	Tumblr.com	✓	✗	46	Thejournal.ie	✗	✗
22	Pinterest.com	✓	✗	47	Eircom.net	✗	✗
23	Imdb.com	✓	✗	48	Sky.com	✗	✗
24	Irishtimes.com	✗	✗	49	Ask.com	✗	✗
25	Daft.ie	✗	✗	50	<omitted>	✗	✗

Table 1 – OAuth Analysis of Ireland's 50 Most Popular Websites [33]

The security evaluation was carried out in a legal and ethical manner and conformed to current Irish and UK legislation. Tools used included the Firefox browser, extensions FireBug and NoRedirect, the HTTP request monitoring application Fiddler2 and command line cURL. The vulnerabilities could only be exposed in the Firefox browser. This was because extensions for stopping HTTP redirects were not available in other browsers. The exploits are due to server side code and therefore independent from the browser. The vulnerabilities should also be demonstrable using command line cURL but this would be quite tedious due to the number of necessary HTTP requests and the size of the each query string or POST body.

3.1.1 Popular Online News Site 1

The sites identified with vulnerabilities were coincidently both online publications. The first exploit was found on one of the leading news sites online who implement social login with Facebook, Google and Twitter. The evaluation was only performed on online accounts that were created for testing purposes. No third party accounts were compromised. Users can register traditionally or via a social network. Once registered, users can comment on articles, provide profile information and subscribe to newsletters. The online user accounts can be hijacked via an OAuth exploit and Cross Site Request Forgery (CSRF) similar to that mentioned in the Known Exploits section. At the time of writing, this online news site was in the 40 most popular websites in Ireland, top 20 in the UK and 150 globally. These statistics however did tend to fluctuate.

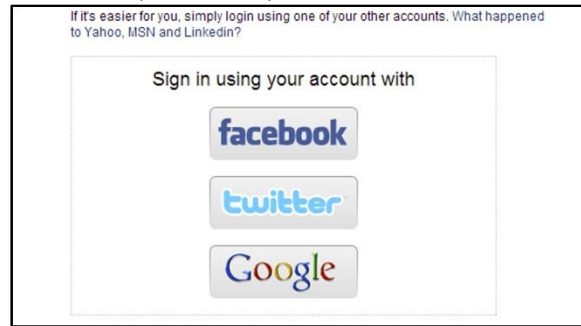


Figure 2 – Popular News Site 1 Login Screen

This is not a man in the middle attack and does not require any knowledge of the underlying system, just a basic understanding of the OAuth process. An attacker could attempt to login via Facebook and use a browser extension, such as NoRedirect, to prevent Facebook from redirecting back to the site with an authorisation code. Aborting the redirect stops the code from being used and therefore still valid. The attacker could then put the online redirect URL with the authorisation code in an iframe, image source tag or other CSRF method in a web page and then send the link to a target victim. If the target is logged into the news site online when opening the link they will have inadvertently linked the attacker's Facebook account with their news site online account as the authorisation code in the redirect URL is associated with the attacker's Facebook profile.

As there is a valid news site online browser session, when the client browser downloads the web page, the request to the redirect URL will be executed with the cookies that have been set for that domain. If the redirect URL is set to an image source attribute then the request will be transparent to the victim. The attacker would then be able to log in using his Facebook credentials and have access to the victims online account. Sensitive information such as date of birth, email addresses, location details plus the ability to comment on articles will be exposed to the attacker. The attacker could also analyse the victim's newsletter subscriptions to build a more targeted socially engineered attack. This attack does rely on the victim having an unexpired Online session when opening the malicious link but due to the popularity of the web site this attack could potentially have a high success rate. Consider an attacker, Mallet, who wishes to hijack online accounts. Mallet creates an account with the news site traditionally. He then attempts to link his Facebook account to his Online account via the option in the Profile Preferences. Upon signing into Facebook Mallet is then presented with the access consent dialog below in Figure 3.

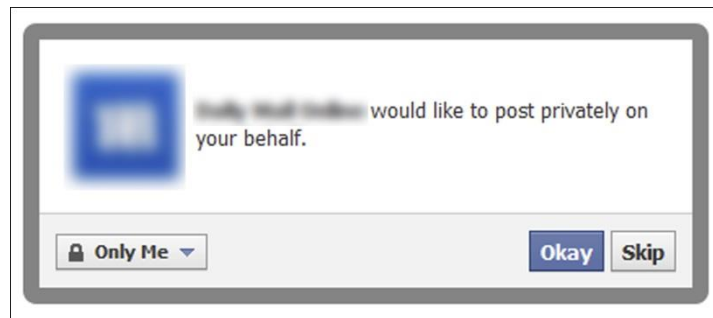


Figure 3 - Facebook OAuth Consent Dialog

Before clicking "Okay", Mallet disables all redirects in the NoRedirect options window by unchecking the allow checkbox for a regex pattern (".*") which covers all URLs. Mallet is then shown the redirect URL with an authorisation code which wasn't executed as Figure 4 demonstrates.

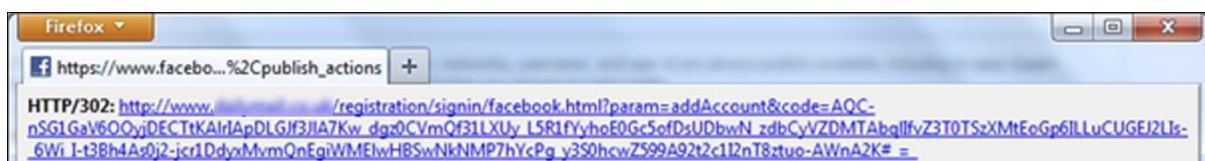


Figure 4 - Stopped Redirect with Authorisation Code

Mallet then creates a malicious web page with an image tag that has the source attribute pointing to the redirect

Please cite as: Kevin Gibbons, John O'Raw, Kevin Curran (2014) Security Evaluation of the OAuth 2.0 Framework. Information Management and Computer Security, Vol. 22, No. 3, December 2014, ISSN: 0968-5227
 URL. The link to this web page is then put in an email and styled to lure the using into clicking it, such as “Win a Free iPad” or similar incentive.

```
...

```

Listing 1 – Image Tag in Malicious Web Page

Mallet could send this email to masses with the hope that they are currently logged in when opening the link or target the email to a specific few via social engineering. If a user falls for the bait and opens the link they will have unknowingly associated Mallet’s Facebook account to their online profile. Mallet then periodically logs in to the Online account with his Facebook credentials to verify if his attack has succeeded. Figure 5 shows the sensitive information (though some of it is public) that is exposed to Mallet upon a successful hijack.

Figure 5 – Successful Hijacking of online Account

Figure 6 shows the commenting functionality exposed to the attacker.

Figure 6 - Attacker has Ability to Comment on Articles

This attack can be prevented simply by implementing the OAuth state parameter when requesting access, per the specification. The state parameter, typically a nonce that is sent to the authorisation server should also be returned on the redirect URL. The application must verify that the state parameter received is equal to the one that was sent. This binds the access request to the redirect and stops authorisation codes that were issued for

Please cite as: Kevin Gibbons, John O'Raw, Kevin Curran (2014) Security Evaluation of the OAuth 2.0 Framework.

Information Management and Computer Security, Vol. 22, No. 3, December 2014, ISSN: ISSN: 0968-5227

other grants being used with other redirects. This vulnerability is also listed in Section 4.4.1.8. of the OAuth 2.0 Threat Model. Listing 2 shows an OAuth request URL with the state parameter and the corresponding redirect URL with the same state parameter.

```
Request:https://www.facebook.com/dialog/oauth?client_id=...&redirect_uri=https%3A%2F%2Fwww.
adverts.ie%2Flogin%2Ffacebook&state=462395a3d9a1f5c56eae33d494962762&scope=email

Response:
https://www.adverts.ie/login/facebook?code=AQC7HnKgEYrlwKrZHpFsFzFz2aBzmcjPgTeT62mriv2y6pL
egkqMvGO9M6NtXt7yR-
p80EERj_CCSVHNUgvZ6kJvu2P0bIkt3VDrjjs_mn9Ua5jqUn2hKYEp5sAyx89fdxtcNWRZZkcZdaQsMrDondxVfgRf
wAU9QFFAQWVdTPSG0ScrPLlMIb7O7H1NLt1K_5P*xjKv9ijkGC8IggNUW_oL&state=462395a3d9a1f5c56eae33d
494962762#_=_
```

Listing 2 - OAuth Implementation with State Parameter

This vulnerability was reported to site but at the time of writing no acknowledgement had been received.

3.1.2 Popular News Site 2

Another leading news site was susceptible to exactly the same attack, though less impacting. The resulting linkage between the attacker's Facebook to the victim's online account overwrites most of the victim's personal information with that of the attacker's. However, a full address was left exposed as well as commenting functionality. The results of this attack were inconclusive as it was not always successful. They were not informed as a vulnerability could not be proved. It is possible that they are working on their implementation and that the issue was resolved during the evaluation. The varying results could also be due to a missing step or URL parameter that was not realised when the attack did succeed.

3.1.3 Popular News Site 3

Though not in Ireland's top 50 websites, a leading US online news site was also found to expose a security vulnerability. After logging in to the AOL owned site via Facebook, the access token is stored in a browser cookie. This is essentially the same as saving the user's Facebook credentials on the machine. The OAuth 2.0 Bearer Token specification states that access tokens must not be stored in cookies [36]. An attacker could deploy cookie harvesting techniques (via botnet or physical access to the device) to gain the access token which he could then use on the Facebook Graph API to retrieve sensitive information about the user and all the user's friends. For the access token to remain valid, the user must have a current Facebook session. The cookie is also not reset upon logging out of the site.

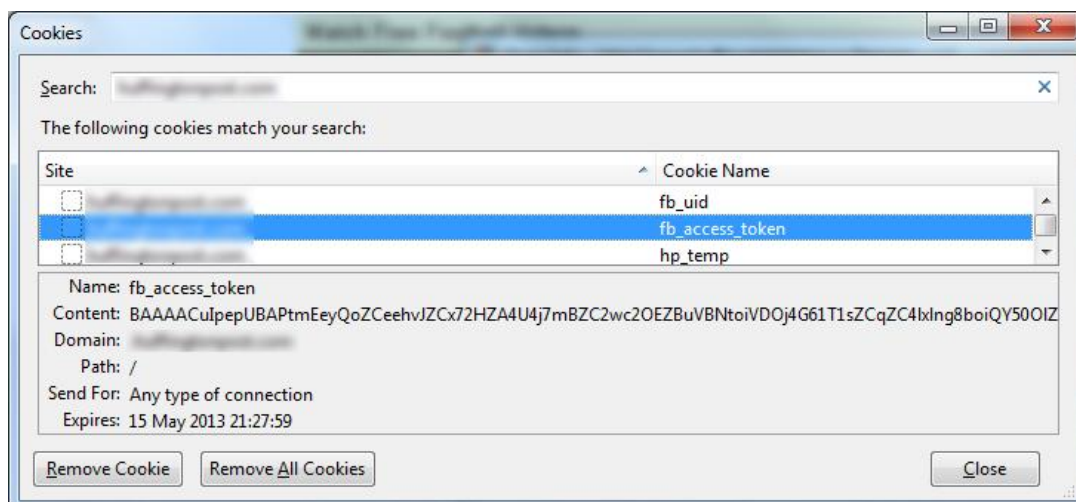


Figure 7 - Facebook Access Token Stored in Cookie

Figure 8 shows the JSON response an attacker could potentially get when querying the Facebook resource server with the stolen access token. The ids in the "friends" object are links to the friends' personal details that were requested in the URL.

```
Request:

https://graph.facebook.com/1143124064?fields=id,name,address,birthday,location,
friends.fields%28address,birthday,email,location%29,email
&access_token=BAAAACuIpepUBAptmEeyQoZCeehvJ...

{
  "id": "1143124064",
  "name": "Kevin Gibbons",
  "birthday": "08/25/1987",
  "location": {
    "id": "113114048705673",
    "name": "Milford, Donegal, Ireland"
  },
  "email": "kevingibbons2007@hotmail.com",
}
```

Figure 8 - Facebook Graph API Response

The site was in the 20 most visited website in the US and was ranked in top 100 globally at the time of writing [37]. This vulnerability can be mitigated by saving the access token in secure online storage and reducing the lifetime of the token. The site was informed of this issue and advised that they would evaluate the matter and take the appropriate actions. Shortly after the vulnerability was resolved, the access token is no longer stored in a browser cookie.

4. OAuth Framework Implementation

To test the robustness of the OAuth framework an entire OAuth environment was required. The proposed solution would compose of three parts: a client application, an authorisation server and a resource server. The client application needed to consume OAuth enabled services. The authorisation server had to manage access to the resource server. The resource server had to expose data from the database based on the authorisation the user would be given from the authorisation server. It was decided that the client application would consume emails from Google's Gmail API. The authorisation and resource server were modelled around a basic task tracking web application. The client application would also consume task data from the developed resource server. The client application would also support Single Sign On for Google and Facebook as well as a developed identity provider "MyTasks". The authorisation server delegated authorisation to the client application and stored cryptography information for each access grant. The resource server validated the supplied access token via public cryptography and returned the requested data.

4.1 Design

The use case diagram demonstrates the actions the user can perform in the client application based on the requirements in the previous section.

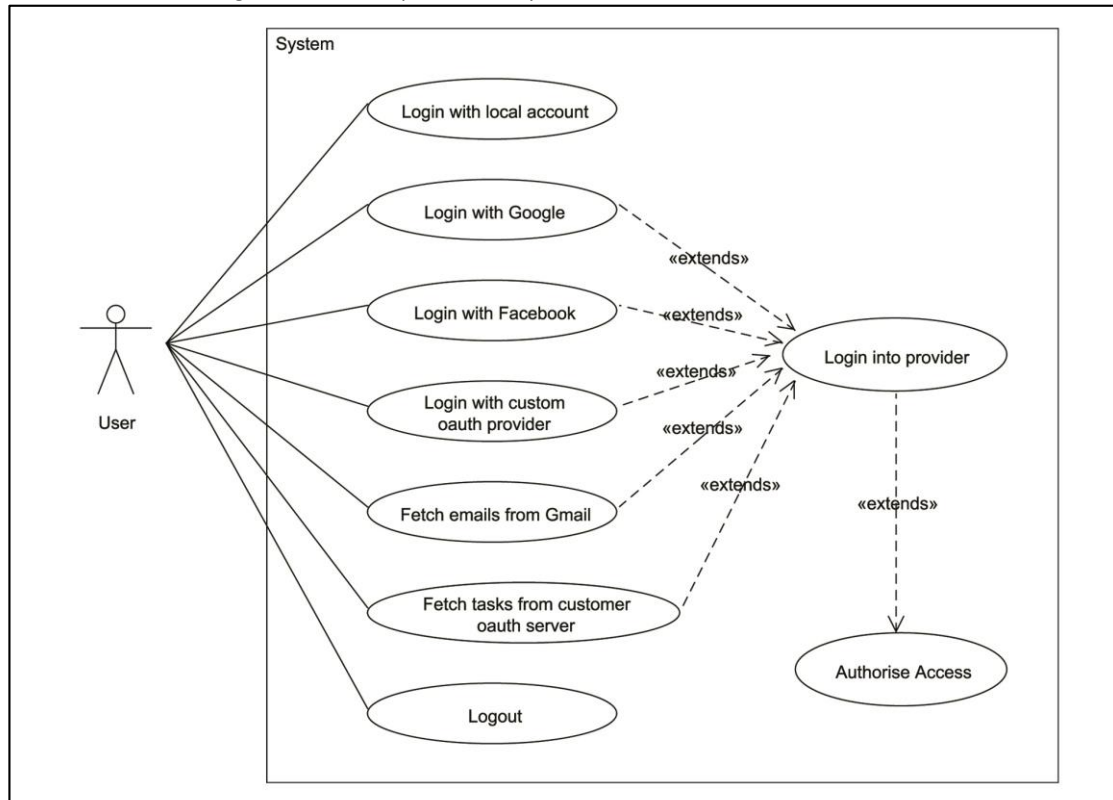


Figure 9 - OAuth Server Side Client Use Case Diagram

Some of the actions, such as “Login with Facebook”, will invoke the OAuth provider to authenticate the user and confirm data access consent.

The use case diagram in Figure 10 shows the actions that the user and client application can perform in the authorisation server. The user will be able to login in and out of the web application as well as grant and revoke third party applications to their data. A client application will be able to perform standard OAuth actions. For brevity the Query API action has been included in the authorisation server though in practise this is actually happens in the resource server.

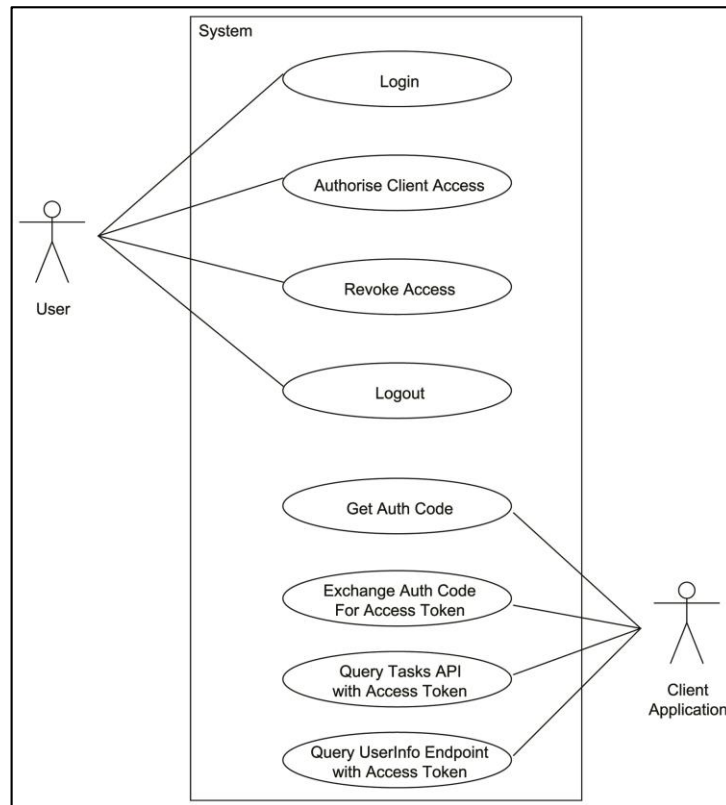


Figure 10 – OAuth Authorisation Server Use Case Diagram

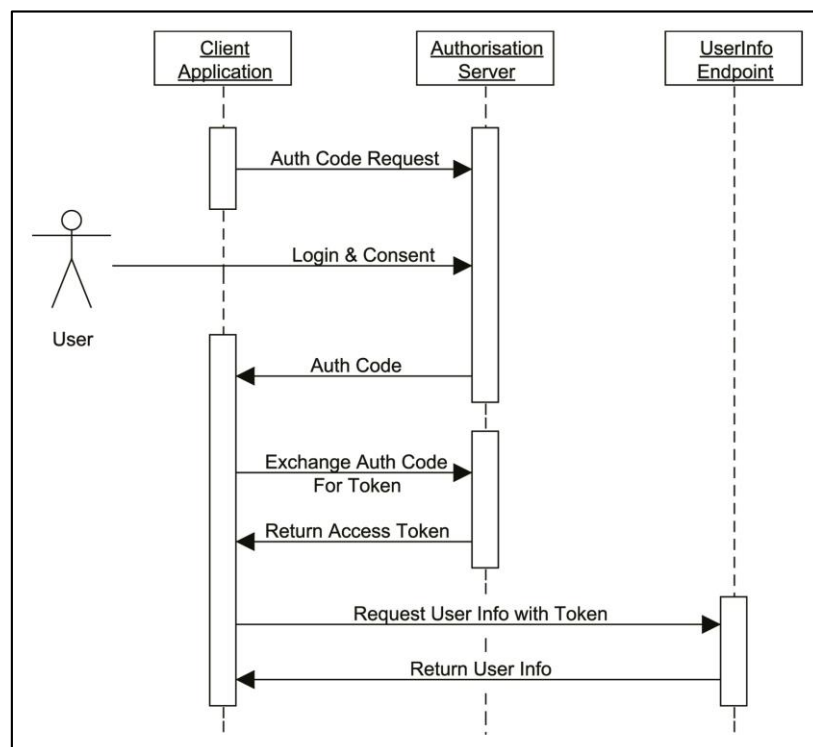


Figure 11 – OAuth SSO Sequence Diagram

The sequence diagram above in Figure 11 illustrates the lifetime of each endpoint in the OAuth Single Sign On process. The client application is the main component in this sequence. The process is invoked by the client application when it requests an authorisation code from the authorisation server. As the client web application is stateless, its lifetime ceases when the browser redirects to the authorisation server. Upon user consent, the

authorisation server redirects back to the client application. As the authorisation server is a stateless HTTPS end point its lifetime ends. The client then makes another request to the authorisation server to exchange the authorisation code for an access token. This request is asynchronously, not a browser redirect, hence the lifetime of the client application has not ended. When the client receives an access token it makes another asynchronous request to the resource server to fetch user information.

Per the OAuth 2.0 specification, the authorisation server has to persist the registered client applications and authorisation grants. The entity relationship model in Figure 12 shows the tables required to facilitate OAuth and federate user information. The user table stored user information. Following best database security practices, user passwords were salted and hashed. The client table stored client application information which was used for validating authorisation requests. The client authorisation table stored access grant information. The nonce table was for storing one time authorisation codes. An authorisation code has to be a unique random value. This is guaranteed by the nonce table. The CryptoKey table stores the symmetric keys used to encrypt and sign authorisation codes and refresh tokens.

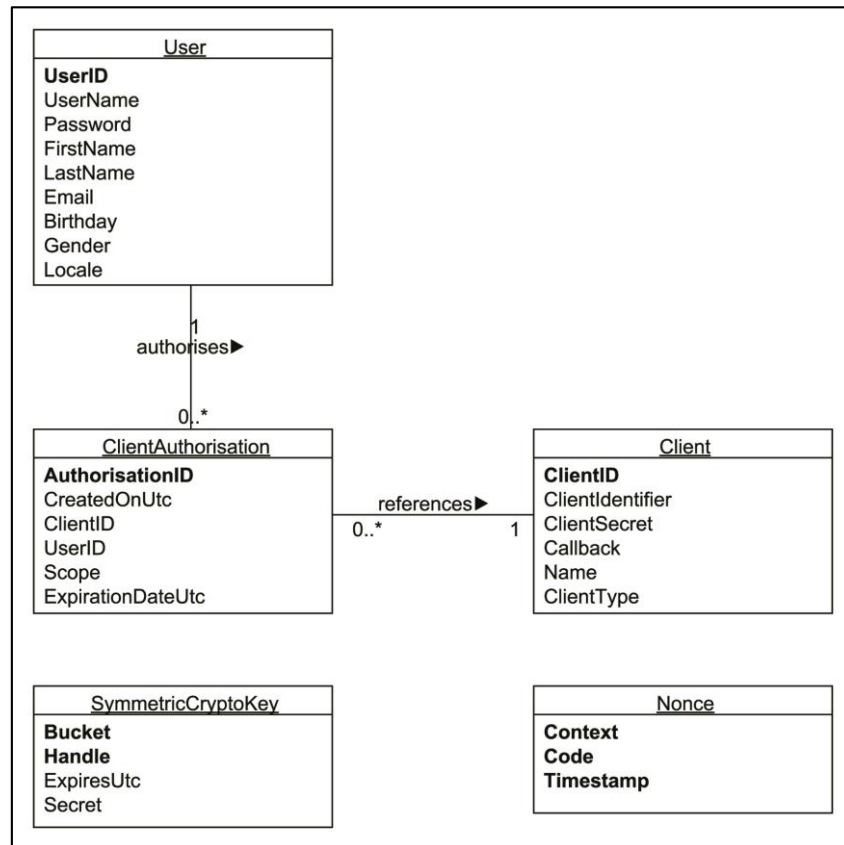
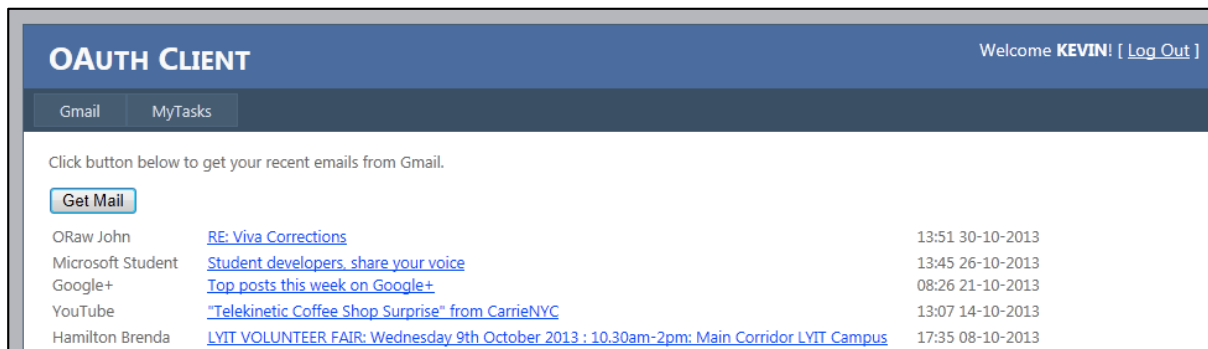


Figure 12 - Authorisation Server Database Entity Relationship Model

4.2 Implementation

The OAuth environment was developed in C#. The solution was developed on a machine running Windows 7 though it was deployed on a production grade server OS, such as Window Server 2008 or 2012 if deployed. Microsoft SQL Server [24] was the obvious database choice due to the tight integration with .NET languages, stability and previous experience. The client and servers were deployed to Microsoft's IIS (Internet Information Services) server. A third party DLL called DotNetOpenAuth (DNOA) was used to implement the OAuth interactions. DNOA is an open source .NET library for creating OAuth enabled APIs which abstracts most of the cryptography and complexity. DNOA provides skeleton classes and database table structure specifications for each OAuth entity. Two databases were created, one for the client application and one which was shared between the authorisation server and the resource server. This server database consisted of tables and columns that DNOA required plus some custom tables and attributes to store user and task data.

The first stage of development focused solely on creating a client application which consumed a third party API protected by OAuth. Google's Gmail API was selected due to their verbose OAuth authentication documentation. Before any coding, the OAuth sequence that would be carried out by the client was simulated in the Google OAuth Playground. This gave a clear overview of each step in the authentication process and the email data that would be returned. A registration in the Google Code Console was then made to obtain a client ID and secret key for the client application. A basic ASP.NET web application was then developed which supported login via the standard ASP.NET login control for ASP.NET Membership which is integrated with SQL Server. An aspx page was created to display a list of Gmail messages. An Email object was created to hold the data returned from the web service. A combination of XDocument and LINQ was used to parse the XML response, iterate through the nodes and project the data into an email entity. A string representing a HTML table to display the data was then built. Figure 13 shows the list of emails returned by Google. Clicking the subject opened another browser tab displaying the email message in Gmail. Though CSS was used throughout development it was not a primary focus.

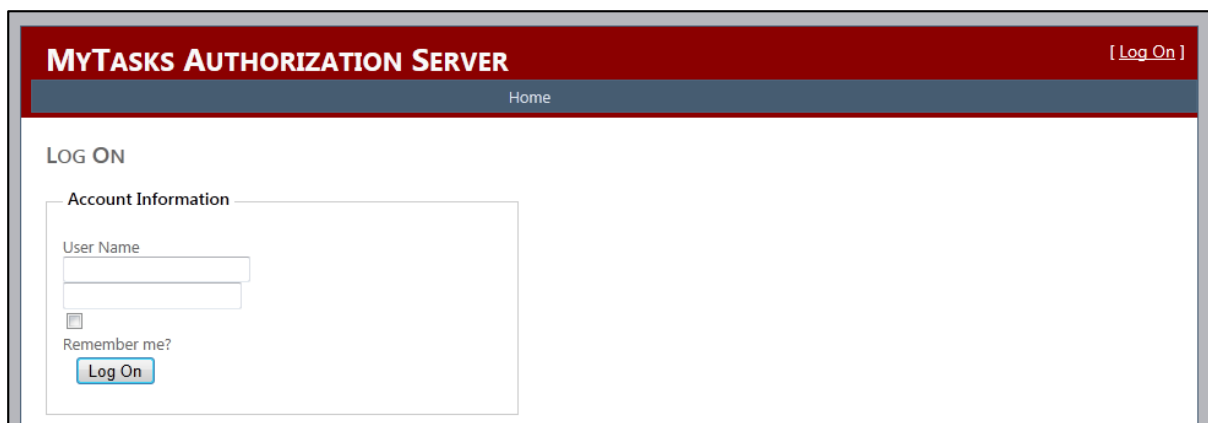


The screenshot shows a web application titled "OAUTH CLIENT". In the top right corner, it says "Welcome KEVIN! [Log Out]". Below the title bar, there are two tabs: "Gmail" (selected) and "MyTasks". The main content area has a heading "Click button below to get your recent emails from Gmail." followed by a "Get Mail" button. Below the button is a table of email data:

ORaw John	RE: Viva Corrections	13:51 30-10-2013
Microsoft Student	Student developers share your voice	13:45 26-10-2013
Google+	Top posts this week on Google+	08:26 21-10-2013
YouTube	"Telekinetic Coffee Shop Surprise" from CarrieNYC	13:07 14-10-2013
Hamilton Brenda	LYIT VOLUNTEER FAIR: Wednesday 9th October 2013 : 10.30am-2pm: Main Corridor LYIT Campus	17:35 08-10-2013

Figure 13 - OAuth Gmail Client

The next stage in development was to expand the client application to consume a custom OAuth enabled API. To do this, an authorisation server and resource server were created. The authorisation server and resource server were created based on sample code and guidelines provided by DNOA. The authorisation server was an MVC web application. The necessary tables were created in the database and then the skeleton classes were expanded with custom code. The tables were mapped to objects using .NET's LINQ to SQL framework. This had many benefits compared to writing custom SQL queries, classes and transformations including reduced development time, concise code, built in filter methods and security. Initially the client application failed to get task data from the resource server. DNOA threw an exception as the client was not running on HTTPS. A self-signed SSL certificate had to be generated in IIS and applied to the website bindings. The authorisation server and resource server web configs were then modified to reflect the new transport settings.



The screenshot shows a web application titled "MYTASKS AUTHORIZATION SERVER". In the top right corner, there is a "[Log On]" link. Below the title bar, there is a "Home" link. The main content area has a heading "LOG ON". Underneath, there is a section titled "Account Information" which contains a "User Name" label, a text input field, a password input field, a "Remember me?" checkbox, and a "Log On" button.

Figure 14 - My Tasks Login

The My Tasks server did not offer a front-end interface for registering client applications. Instead values, such as client key, client secret and redirect URL, were manually inserted in the client database table. Though this was listed in the requirements, a front end for this functionality was not a primary focus of development

and was put a side for a later date but was never completed due to time constraints. Once the token was added to the request and the resource server was able to extract information from it, the task data for the user was returned. The list of task entities was then iterated and a HTML string was built to present the data in a list. CSS was added appropriately to format the list and distinguish between completed and unfinished tasks.



Figure 15 - My Tasks OAuth Client

At this point, a mechanism for offline access had not been implemented. The client application had no way of remembering authorisations. If a user had granted the client access to their tasks and then navigated away from the tasks page the next time they loaded tasks they would have re-authorise the client application. The authorisation server did return a refresh token but it was not being used. For differentiation, it was decided to only implement offline access in the Tasks client. Logic was added to store the access token and refresh token for an authorisation in the client database. LINQ to SQL was again used to map the authorisations table to an object and automatically generate logic for performing CRUD operations. The Tasks client was remodelled to check whether an authorisation already existed and if it had not expired. A method was added to the wrapper class which exchanged the refresh token for another access token. The new access token and refresh token were then updated in the database. The authorisation server was configured to mint access tokens with a lifetime of 2 minutes. This could have been increased which would have reduced requests and improved performance but as the access tokens were stored in the database it would be more secure to give them a short lifetime in case the database was compromised.

The last stage of development was to implement Single Sign On. Initially, adding support to login with identity providers Google and Facebook was developed first then a custom identity provider was created. A button to login with Google was added to the client login page which started the OAuth process using the helper class. As with the other OAuth interactions on the Gmail and Tasks pages, a request was made for an access token after the user had given consent and an authorization code is on the URL query string. Google also returns an `id_token` attribute in the JSON response when requesting an access token with scope for user information. This is a JWT which contains information about the user and the requested authorisation. Per the OpenID Connect specification [17], Google states that the `id_token` must be validated at the token info endpoint. At the time of development, there was no method to extract the `id_token` attribute from the response as it was abstracted in DNOA. A mechanism to extract custom attributes from the authorisation response has been raised as an issue in the library's GitHub repository. Instead the access token was validated at the token info endpoint. If the access token is valid it is passed to the user info endpoint which returns the user information that was signed and encrypted in the JWT `id_token`.

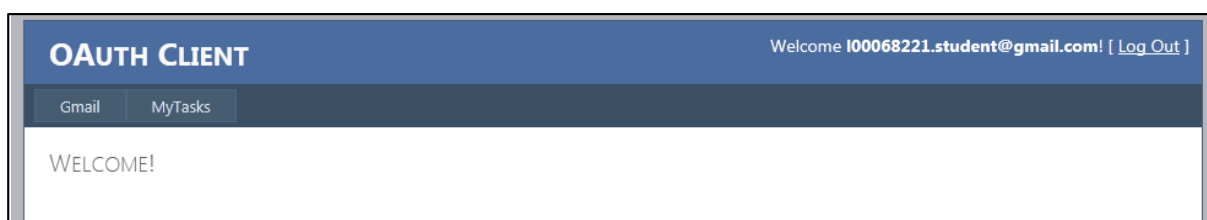


Figure 16 – Client Displaying User Information Federated by Google

The user information is then parsed and mapped to a ProviderUser object and a new session is created via the FormsAuthentication.SetAuthCookie() method. The email address of the user is passed into the method as the identity. This is outputted in the loginview control in the top right corner just as the username would be if the user had logged in locally. In the default page, logic was added to display some of the user data. This can be seen in Figure 16. The next step in development was to create an identity provider that could federate a user's identity to client applications. The resource server was expanded to include a user info endpoint to return information about the user. Extra columns were added to the users table in the server database to hold additional user information such as gender and locale. The user info web service method returned information about the user in JSON, similar to that of Facebook and Google, per the OpenID Connect specification [17].

The login page in the client was then modified to support logging in with this custom identity provider which was also branded as "My Tasks". Logic for logging in with My Tasks followed a similar pattern as the other identity providers except the web service reference client was used instead of a standard web request. The access token was added to the call manually as with the other service calls in the My Tasks client. The user information was then parsed using the same method as the other providers as the data returned was mostly the same format. This method also instigates a new ASP.NET user session.

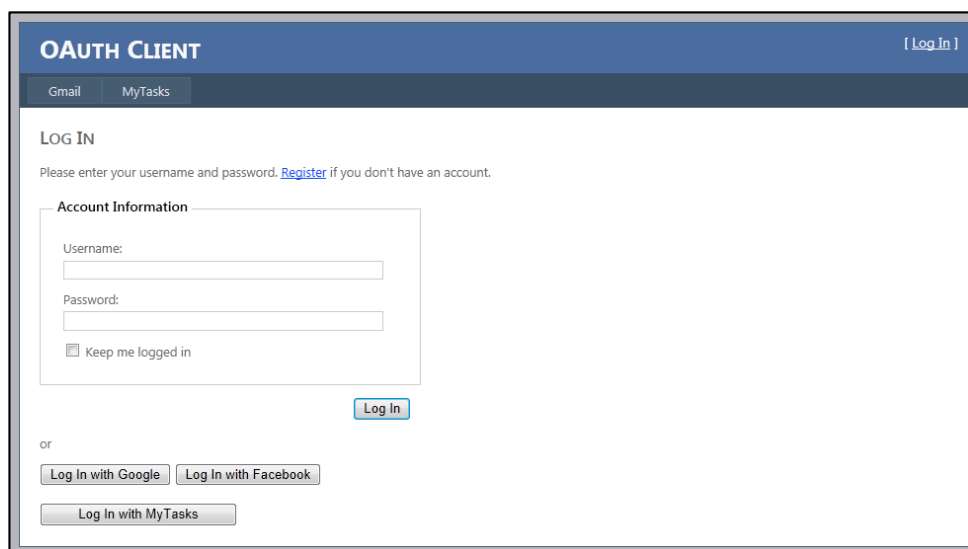


Figure 17 - OAuth Client Login Page with SSO Options

Though the functionality for logging in with My Tasks worked correctly it did not follow the OpenID Connect specification. When user information is requested, the authorisation server should return a JWT id_token as well as an access token, as Google does. This could not be implemented however due to the abstraction that DNOA provides in the token endpoint of the authorisation server. This is why a token info endpoint was not developed in the resource server as there was no id token to validate. Initially, when the My Tasks service was developed, the action URLs for each operation had a default base address of "http://tempuri.org". As the operation action was used to represent scope and was displayed to the user confirming consent it was decided that this should be given an appropriate label. This was achieved by adding a namespace attribute to both the service contract interface and the service class. A binding namespace attribute was also added to the service endpoint configuration in the resource server web config.

5. Evaluation

We focused on the OAuth security and correctness. Though some general security testing was executed protecting the solution from all known web application attacks was not a main focus. The solution was secured against SQL injection attacks through the LINQ to SQL framework as it implements SQL parameters. XSS (Cross Site Scripting) was not considered as the only inputs throughout the solution are usernames, passwords and the OAuth parameters which are all validated. Brute force attacks were also not considered. The developed solution was tested against the threats identified in the threat model as the evaluated industry sites were. Some of the threats could not be verified as they could not be applied to the solution, such as password grant threats and

Please cite as: Kevin Gibbons, John O'Raw, Kevin Curran (2014) Security Evaluation of the OAuth 2.0 Framework.

Information Management and Computer Security, Vol. 22, No. 3, December 2014, ISSN: 0968-5227

attacks from internal resources. If the solution was to be used in industry more preventive measures would have to put in place. A table listing some of the potential vulnerabilities that were tested against the artefact can be seen in Table 2. Tools used included Firefox extensions FireBug and NoRedirect, the HTTP request monitoring application Fiddler2 and command line cURL.

	Threat	Secured	Reason
Clients			
1	Obtaining Client Secrets	✓	Web config isn't accessible from client front end
2	Obtaining Refresh Tokens	✓	DNOA validates client ID
3	Obtaining Access Tokens	✓	Secured in database
Authorisation Endpoint			
4	Password Phishing by Counterfeit AS	✓	Uses HTTPS, educate user about phishing
5	User Unintentionally Grants Access Scope	✓	N/A. Multi-level scope wasn't supported
6	Malicious Client get Existing Authorization	✓	DNOA authorises client before returning auth code
7	Open Redirector	N/A	N/A
Token Endpoint			
8	Eavesdropping Access Tokens	✓	Uses HTTPS. Tokens have a short lifetime
9	Obtaining Access Tokens AS database	✓	MS SQL is secured by password. Secured from SQL injection
10	Disclosure of Client Credentials	✓	Uses HTTPS
11	Obtaining Client Secret from AS Database	✓	MS SQL is secured by password. Secured from SQL injection
12	Obtaining Client Secret by Online Guessing	✓	High entropy secrets.
Authorisation Code			
13	Eavesdropping or Leaking Authoriz'n "codes"	✓	Uses HTTPS, Nonce table ensures one time use.
14	Obtaining "codes" from AS Database	✓	MS SQL is secured by password. Secured from SQL injection
15	Online Guessing of Authorization "codes"	✓	DNOA validates client and redirect URL
16	Malicious Client Obtains Authorization	✓	DNOA validates client and redirect URL
17	Authorization "code" Phishing	✓	Uses HTTPS
18	User Session Impersonation	✓	Uses HTTPS
19	Autho'n Leakage through Counterfeit Client	✓	Redirect URL is validated
20	CSRF Attack against redirect-uri	✓	DNOA uses state param to bind requests and responses
21	Clickjacking Attack against Authorization	✓	AS adds x-frame-options header to the authorise endpoint
22	Resource Owner Impersonation	✗	Would need to deploy multi factor authentication
23	DoS Attacks That Exhaust Resources	✗	Would need to add logic to authorisation server
24	DoS Using Manufactured Author'n "codes"	✓	DNOA uses state param
25	Code Substitution (OAuth Login)	✓	Codes are bound to client IDs

Table 2: Artefact Threat Model Partial Results

The solution proved not to be vulnerable to the majority of the listed threats. As Table 2 demonstrates, many of the issues identified in the threat model were mitigated by deploying end to end encryption. The artefact was however not protected from Threat 22 – Resource Impersonation. This is where a malicious client application programmatically completes the entire OAuth server side flow via POST requests without the user's knowledge. The user must have a valid session with the OAuth provider to carry out this attack. To secure the solution from this attack, a CAPTCHA could be added to authorisation server consent dialog. Another option suggested in the threat model is to notify the user (email or SMS) upon each authorisation grant. Though trivial to apply, these recommendations were out of the scope of research. The solution was also vulnerable to Threat 23 – DoS Attacks on the Resource Server, but as previously mentioned, preventing denial of service attacks were not a main focus, though an important part of securing web based applications. Several of the threats listed were not applicable to the developed solution. This was because the issues were for less mainstream grant extensions, such as the user credentials flow, that weren't supported or they did not apply to the infrastructure.

When testing each threat against the artefact it was discovered that it was initially susceptible to Threats 17, 18 and 19. Threat 17 and 18 involve a malicious application mocking a genuine application to phish for authorisation codes and access tokens. DNS spoofing would be required for this attack. Per the recommendation in the threat model, HTTPS was applied to OAuth client. Although SSL had been applied to the authorisation and resource server the client application was still on HTTP as HTTPS was not a requirement for the client. SSL mitigates this threat as the server certificate contains the correct DNS information. Threat 19 is where an attacker exploits a legitimate client application by invoking an authorisation request and injecting a different redirect URL than

registered with the authorisation server in order to redirect the authorisation code to a malicious client. This threat was prevented by enabling redirect URL validation in DNOA on the authorisation server. Only the base of the host address was checked for equality but this was adequate for this implementation. Though the OAuth side of the solution conformed to the specification, the SSO aspect did not adhere to the OpenID Connect specification. The user information scoped authorisation requests did not return a JWT id token. A token info endpoint to validate the JWT was also not developed. This was due to the lack of OpenID Connect support in the DNOA library and the abstraction it provided which stopped custom data being added to responses. Validating the id token is mandatory to ensure that the authorisation was issued by and targeted to the correct parties as well as validating timestamps. The chance of replay attacks were mitigated though as only the server side grant flow was implemented. At the time of writing, a request to include OpenID Connect support in DNOA was still unassigned. Facebook's user information API also does not conform to the OpenID Connect specification as it does not provide an id token or token validation endpoint.

6. Conclusion

Overall, the developed solution performed well. A high level of security was achieved. The solution conforms to the OAuth 2.0 specification though there were some shortcomings when meeting the requirements of the OpenID Connect specification. The development of the client applications was relatively more straightforward than the authorisation and resource server. The use of the third party library DNOA helped simplify the process. Most major OAuth vendors provide their own client libraries for consuming their services. Though this was an option for providing SSO with Google and Facebook the use of one generic library was better suited to the requirements of the entire solution. DotNetOpenAuth (DNOA) is a comprehensive library for building a C# authorisation server. Developing an authorisation server from scratch would have included complex cryptography and validation. DNOA removed this redundancy. Though limited in places, DNOA provided a solid infrastructure to build the authorisation server on. Documentation for DNOA was poor however. Development relied on assumptions and trial and error. At the time of development, there were few resources on implementing the library for the 2.0 specification. Ironically, after the servers were built, [41] was published, which features a chapter on creating an authorisation server using DNOA. Though the solution was complete at this stage the book was used to confirm that the library had been used correctly. The developed artefact could have been expanded in many areas but this was not possible due to the time constraints. Only the mainstream server side flow grant was implemented. The client side flow and SAML assertion flow would have been interesting grants to develop though this would have contributed to the complexity of the solution. Many large corporations utilise SAML for authenticating their employees. Extending the solution to support logging in via SAML tokens from cloud identity providers, such as Ping Identity, would have greatly added to the value of the artefact.

The environment could have also been improved by adding support for finer grained request scope such as breaking down the "UserInfo" scope into separate fields such as email and birthday. The authorisation and resource servers contained hard coded public and private key pairs. This was sufficient for local testing but a deployed solution would require the key pairs to be derived from X.509 certificates residing on the servers. The authorisation server did not offer a front end to register client applications. It also did not have a front end to let users revoke client applications access to their data. These GUIs were not required to test the OAuth functionality but would be necessary in a deployed environment. Future work may include contributing to the DNOA open source project in GitHub. Adding support for the OpenID Connect specification would benefit the solution and other DNOA users. When implementing OAuth, developers need to decide which support library best suits their requirements. Client applications may not need the support of a library but they do offer built in security and help simplify the process. If only one OAuth vendor, such as Facebook, is going to be consumed by a client application then it may be best to use Facebook's own client library rather than a generic library. This would eliminate the need to cater for Facebook inconsistencies when using a general library. It also passes the onus of security onto the vendor.

Major vendors, such as Flickr and Twitter, have invested heavily in the OAuth 1.0 specification. It does offer high security through its encrypted and signed messages but its complexity does make it difficult to integrate with. As more vendors implement OAuth 2.0 support for 1.0 will likely decrease. To create a rich set of universally integratable web applications vendors need to align their OAuth implementations to ensure all round compatibility. As we interact with more data on our smartphones and tablets, in the future, it is likely that an OAuth consent dialog will be built into mobile devices and become more present. Google's Android OS currently

offers a native OAuth dialog for confirming access to their own APIs but a generic OAuth consent UI will likely be developed. It is predicted that Apple's iOS will follow suit. As OAuth grants become more ubiquitous on the web, it is likely that two step authentication will be incorporated into grant consent dialogs to add another layer of security by authenticating users from an unfamiliar location with SMS codes. As SSO becomes more widespread, it is likely that future internet users will only have one main account or "presence" in the web. Google, Facebook and Microsoft are strong contenders in the identity provider race. A disadvantage of this sole account is that it provides one point of failure for attack. If a user's Google account was compromised then all the linked accounts could also be compromised. This is why multi factor authentication will be a necessity in future online account access.

The Dynamic Client Registration specification will likely improve the whole OAuth integration process. Giving a client application the ability to register with an OAuth provider on the fly without any previous configuration or developer intervention will greatly contribute to the interoperability of web applications.

Throughout the research, Google's OAuth documentation has been an invaluable resource. The Google Playground has also been a very useful tool. The Google Playground web application allows developers to test Google's OAuth protected APIs. It helped demystify the abstracted sequence of events that occur in an authorisation grant. It steps through each request and response and lets the developer see how each operation is choreographed. An initially complex process like OAuth is best understood when broken down and demonstrated visually in this manner. The OAuth framework has many advantages over traditional credential authentication. It also opens up vast opportunities for integration between applications. As with any bleeding edge technology, there are drawbacks and security concerns that need to be addressed. The developed solution proves that a reasonably secure OAuth environment can be achieved. The artefact was secured from almost all the known threats that were applicable. Reaching this level of protection took time. The OAuth process is complex. The various parameters in the requests and responses allow for multiple entry points which need secured. Due to the recent popularity of social media, integrating applications with social networks has become a sort after commodity. Developers often implement this functionality without being aware of the security vulnerabilities. As OAuth is complex and initially daunting, by the time a working authorisation grant has been achieved security aspects have been overlooked. Even the developed OAuth solution contained undetected vulnerabilities that were not identified until threat testing. This is partly due to the abstraction of events that OAuth provides – "*out of sight, out of mind*". OAuth implementations started to appear while the specification was still in draft status. The draft was in early stages with recommendations changing frequently resulting in developers being unsure how to build a secure implementation whilst wanting to be at the forefront of the movement.

OAuth is flexible in that it offers extensions to support varying situations and existing technologies. A disadvantage of this flexibility is that new extensions typically bring new security exploits. Members of the IETF OAuth Working Group are constantly refining the draft specifications and are identifying new threats to the expanding functionality. OAuth provides a flexible authentication mechanism to protect and delegate access to APIs. It solves the password re-use across multiple accounts problem and stops the user from having to disclose their credentials to third parties. Filtering access to information by scope and giving the user the option to revoke access at any point gives the user control of their data. OAuth does raise security concerns, such as defying phishing education, but there are always going to be security issues with any authentication technology. Though several high impacting vulnerabilities were identified in industry, the developed solution proves the predicted hypothesis that a secure OAuth environment can be built when implemented correctly. Developers must conform to the defined specification and are responsible for validating their implementation against the given threat model. OAuth is an evolving authorisation framework. It is still in its infancy and much work needs done in the specification to achieve stricter validation and vendor conformity. Vendor implementations need to become better aligned in order to provide a rich and truly interoperable authorisation mechanism. Once these issues are resolved, OAuth will be on track for becoming the definitive authentication standard on the web.

7. References

1. Honan, M. (2012). *How Apple and Amazon Security Flaws Led to My Epic Hacking*. Available: <http://www.wired.com/gadgetlab/2012/08/apple-amazon-mat-honan-hacking>. Last accessed 4th December 2012.

Please cite as: Kevin Gibbons, John O'Raw, Kevin Curran (2014) Security Evaluation of the OAuth 2.0 Framework. Information Management and Computer Security, Vol. 22, No. 3, December 2014, ISSN: 0968-5227

2. Dinesha, A., Agrawal, K. (2012). Multi-level Authentication Technique for Accessing Cloud Services. *International Conference on Computing, Communication and Applications*.
3. Grosse, E., Upadhyay, M. (2013). Authentication at Scale. *IEEE Security & Privacy*, Vol. 11, No. 1, pp.15-22.
4. Hardt, E., (2012). *The OAuth 2.0 Authorization Framework*, Available: <http://tools.ietf.org/html/rfc6749>. Last accessed 24th November 2012.
5. Leiba, B., (2012). OAuth Web Authorization Protocol. *IEEE INTERNET COMPUTING*. Vol. 16, No. 1, pp.74-77.
6. Connolly, P.J., (2010). OAuth is the 'hottest thing' in identity management. *eWeek*. Vol. 27, Issue 9, pp.12-14.
7. Jannikmeyer, P., (2013). Number News. *Engineering & Technology*, pp.20.
8. Duggan, M., Brenner, J., (2013). The Demographics of Social Media Users — 2012. *Pew Research Center*.
9. Campbell, B., Mortimore, C., Jones, M., Goland, Y., (2013). *Assertion Framework for OAuth 2.0*. Available: <http://tools.ietf.org/html/draft-ietf-oauth-assertions-10>. Last accessed 20th January 2013.
10. Campbell, B., Mortimore, C., (2012). *SAML 2.0 Bearer Assertion Profiles for OAuth 2.0*. Available: <http://tools.ietf.org/html/draft-ietf-oauth-saml2-bearer-15>. Last accessed 20th January 2013.
11. Campbell, B., Mortimore, C., Jones, M., (2012). *JSON Web Token (JWT) Bearer Token Profiles for OAuth 2.0*. Available: <http://tools.ietf.org/html/draft-ietf-oauth-jwt-bearer-04>. Last accessed 20th January 2013.
12. OASIS, (2008). *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. Available: <https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf>. Last accessed 21st January 2013.
13. Sun, S., Beznosov, K., (2012). An Empirical Analysis of OAuth SSO Systems. *2012 ACM Conference on Computer and Communications Security*, pp.378-390.
14. Duggan, G., (2012). Rational Security: Modelling everyday password use. *International Journal of Human-Computer Studies*, pp.415-431.
15. Salesforce, (2012). *SAML Assertion Flow*. Available: http://help.salesforce.com/help/doc/en/remoteaccess_oauth_SAML_bearer_flow.htm. Last Accessed 25th February 2013.
16. Jones, M., Balfanz, D., Bradley, J., Goland, Y., Panzer, J., Sakimura, N., Tarjan, P., (2011). *JSON Web Token (JWT)*. Available: <http://tools.ietf.org/html/draft-jones-json-web-token-10>. Last accessed 21st January 2013.
17. Sakimura, N., Bradley J., Jones, M., de Medeiros, B., Mortimore, C., (2013). *OpenID Connect Basic Client Profile 1.0 - Draft 23*. Available: <http://openid.net/specs/openid-connect-basic-1.0.html>. Last accessed 24th January 2013.
18. Obrenović, Z., den Haak, B., (2012). Integrating User Customization and Authentication: The Identity Crisis. *Security & Privacy*, pp.82-85.
19. Lodderstedt, T., McGloin, M., Hunt, P., (2013). *OAuth 2.0 Threat Model and Security Considerations*. Available: <http://tools.ietf.org/html/rfc6819>. Last accessed 29th March 2013.
20. Richer, J., Bradley, J., Jones, M., Machulak, M., (2013). *OAuth Dynamic Client Registration Protocol*. Available: <http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-07>. Last accessed 3rd April 2013.

21. IETF, (2012). *Web Authorization Protocol (OAuth) - Charter*. Available: <http://datatracker.ietf.org/wg/oauth/charter>. Last Accessed 2nd March 2013.
22. Sakimura, N., Bradley J., Jones, M., (2013). *OpenID Connect Dynamic Client Registration 1.0 - Draft 14*. Available: http://openid.net/specs/openid-connect-registration-1_0.html. Last accessed 3rd April 2013.
23. Cloud Identity, (2011). *OAuth Dynamic Client Registration*. Available: <https://dev.cloudidentity.co.uk/confluence/display/CI/OAuth+Dynamic+Client+Registration>. Last Accessed 2nd March 2013.
24. Richer, J., Mills, W., Tschofenig, H., (2013). *OAuth 2.0 Message Authentication Code (MAC) Tokens*. Available: <http://tools.ietf.org/pdf/draft-ietf-oauth-v2-http-mac-03.pdf>. Last accessed 23rd March 2013.
25. Nouredine, M., Bashroush, R., (2011). A Provisioning Model towards OAuth 2.0 Performance Optimization. *2011 10th IEEE International Conference on Cybernetic Intelligent Systems*, pp.76-80.
26. Shehab, M., Marouf, S., (2012). Recommendation Models for Open Authorization. *IEEE Transactions on Dependable and Secure Computing*, Vol. 9, No. 4, pp.583-595.
27. Browser Auth, (2013). *Channel-Bound Cookies*. Available: <http://www.browserauth.net/channel-bound-cookies>. Last Accessed 4th March 2013.
28. Google Chrome, (2013). *Google Chrome Privacy Whitepaper*. Available: <http://www.google.com/intl/en/chrome/browser/privacy/whitepaper.html>. Last Accessed 14th April 2013.
29. Goma, I., Salama, G., Imam, I., (2012). Biometric OAuth Service based on Finger-Knuckles. *Computer Engineering & Systems 2012*, pp.170-175.
30. Bansal, C., Bhargavan, K., Maffies, S., (2012). Discovering Concrete Attacks on Website Authorization by Formal Analysis. *IEEE 25th Computer Security Foundations Symposium 2012*, pp.247-262.
31. Bansal, C., Bhargavan, K., Delignat-Lavaud, A., Maffeis, S., (2013). Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. *Second International Conference on Principles of Security and Trust – POST 2013*, pp.126-146.
32. Huang, L., Moshchuk, A., Wang, J., Schechter, S., Jackson, C., (2012). Clickjacking: Attacks and Defenses. *21st USENIX Security Symposium*, pp.413-428.
33. Boshmaf, Y., Muslukhov, I., Beznosov, K., Ripeanu, M., (2011). The Socialbot Network: When Bots Socialize for Fame and Money. *2011 Annual Computer Security Applications Conference*, pp.93-102.
34. Boshmaf, Y., Muslukhov, I., Beznosov, K., Ripeanu, M., (2012). Key Challenges in Defending Against Malicious Socialbots. *5th USENIX Conference on Large-Scale Exploits and Emergent Threats – LEET 2012*.
35. Alexa, (2013). *Top Sites by Category: Regional/Europe/Ireland*. Available: <http://www.alexa.com/topsites/category/Top/Regional/Europe/Ireland>. Last Accessed 14th March 2013.
36. Hardt, E., Jones, M., (2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Available: <http://tools.ietf.org/html/rfc6750>. Last accessed 29th March 2013.
37. Alexa, (2013). *Huffingtonpost.com Site Info*. Available: <http://www.alexa.com/siteinfo/huffingtonpost.com>. Last Accessed 16th March 2013.
38. Liu, K., Xu, K., (2012). OAuth Based Authentication and Authorization in Open Telco API. *2012 International Conference on Computer Science and Electronics Engineering*, pp.176-179.

Please cite as: Kevin Gibbons, John O'Raw, Kevin Curran (2014) Security Evaluation of the OAuth 2.0 Framework. Information Management and Computer Security, Vol. 22, No. 3, December 2014, ISSN: 0968-5227

39. Ghazizadeh, E., Zamani, M., Manan, J., Pashang, A., (2012). A Survey on Security Issues of Federated Identity in the Cloud Computing. *2012 IEEE 4th International Conference on Cloud Computing Technology and Science*, pp.562-565.
40. Boyd, R., (2012). *Getting Started with OAuth 2.0*. 1st ed. CA, USA: O'Reilly Media.
41. Lakshmiraghavan, B., (2013). *Pro ASP.NET Web API Security*. 1st ed. USA: Apress.
42. Ping Identity, (2011). *The Essential OAuth Primer: Understanding OAuth for Securing Cloud APIs*. Available: <http://www.innovation-district.com/wp-content/uploads/2012/04/The-Essentials-of-OAuth.pdf>. Last Accessed: 17th December 2012.
43. Google, (2012). *Using OAuth 2.0 to Access Google APIs*. Available: <https://developers.google.com/accounts/docs/OAuth2>. Last Accessed 25th February 2013.
44. Facebook, (2012). *Login for Server-side Apps*. Available: <https://developers.facebook.com/docs/howtos/login/server-side-login/>. Last Accessed 22nd February 2013.
45. Twitter, (2012). *PIN-based authorization*. Available: <https://dev.twitter.com/docs/auth/pin-based-authorization>. Last Accessed 11th February 2013.
46. GoogleDevelopers, (2012). *Google I/O 2012 - OAuth 2.0 for Identity and Data Access*. Available: <http://www.youtube.com/watch?v=YlHyeSuBspl>. Last Accessed: 12th December 2012.
47. Pai, S., Sharma, Y., Kumar, S., Pai, R., Singh, S., (2011). Formal Verification of OAuth 2.0 using Alloy Framework. *IEEE International Conference on Communication Systems and Network Technologies 2011*, pp.655-659.