

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/341336409>

OAuth 2.0: A Framework to Secure the OAuth-Based Service for Packaged Web Application

Chapter · January 2020

DOI: 10.4018/978-1-7998-3355-0.ch005

CITATIONS

0

READS

3,932

5 authors, including:



Shawon S. M. Rahman

University of Hawai'i at Hilo

104 PUBLICATIONS 1,238 CITATIONS

SEE PROFILE



Nazmul Hossain

Jessore University of Science and Technology

26 PUBLICATIONS 79 CITATIONS

SEE PROFILE



Md. Alam Hossain

Jessore University of Science and Technology

51 PUBLICATIONS 288 CITATIONS

SEE PROFILE



Md. Zobayer Hossain

Jessore University of Science and Technology

4 PUBLICATIONS 17 CITATIONS

SEE PROFILE

Chapter 5

OAuth 2.0: A Framework to Secure the OAuth–Based Service for Packaged Web Application

Shawon S. M. Rahman

University of Hawaii at Hilo, USA

Nazmul Hossain


 <https://orcid.org/0000-0002-3743-5193>

*Jashore University of Science and Technology,
Bangladesh*

Md Alam Hossain

*Jashore University of Science and Technology,
Bangladesh*

Md Zobayer Hossain

 <https://orcid.org/0000-0002-5687-3121>

*Jashore University of Science and Technology,
Bangladesh*

Md Hassan Imam Sohag

Jashore University of Science and Technology, Bangladesh

ABSTRACT

OAuth is an open security standard that enables users to provide specific and time-bound rights to an application to access protected user resources. It stored on some external resource servers without needing them to share their credentials with the application. Unlike websites, for locally installed packaged web applications, the main security challenge is to handle the redirect response. The OAuth flow initiated from packaged web apps is similar to the OAuth flows explained in the current literature. However, for packaged web apps, it is difficult to define an HTTP endpoint as redirection endpoint since these apps are locally installed. The authors have proposed a novel method to execute OAuth flow from such applications with the help of a web runtime framework that manages the life cycle of these applications. They have compared their approach with another two existing approaches. After conducting experiments, they have found their approach blocking all illegal OAuth flow executions. The approach also delivers better OAuth response handling time and power consumption performance.

DOI: 10.4018/978-1-7998-3355-0.ch005

OAuth 2.0

INTRODUCTION

OAuth 2.0 is an authorization framework that enables applications to gain restricted access to user accounts on an HTTP service, like as GitHub, Facebook, Twitter, and Digital Ocean. OAuth 2.0 is working by delegating user authentication to the service which is hosting the user account and authorizing 3rd party applications to access user account. OAuth 2.0 gives authorization flows (Hardt Ed., 2012) for desktop and web applications, and mobile devices.

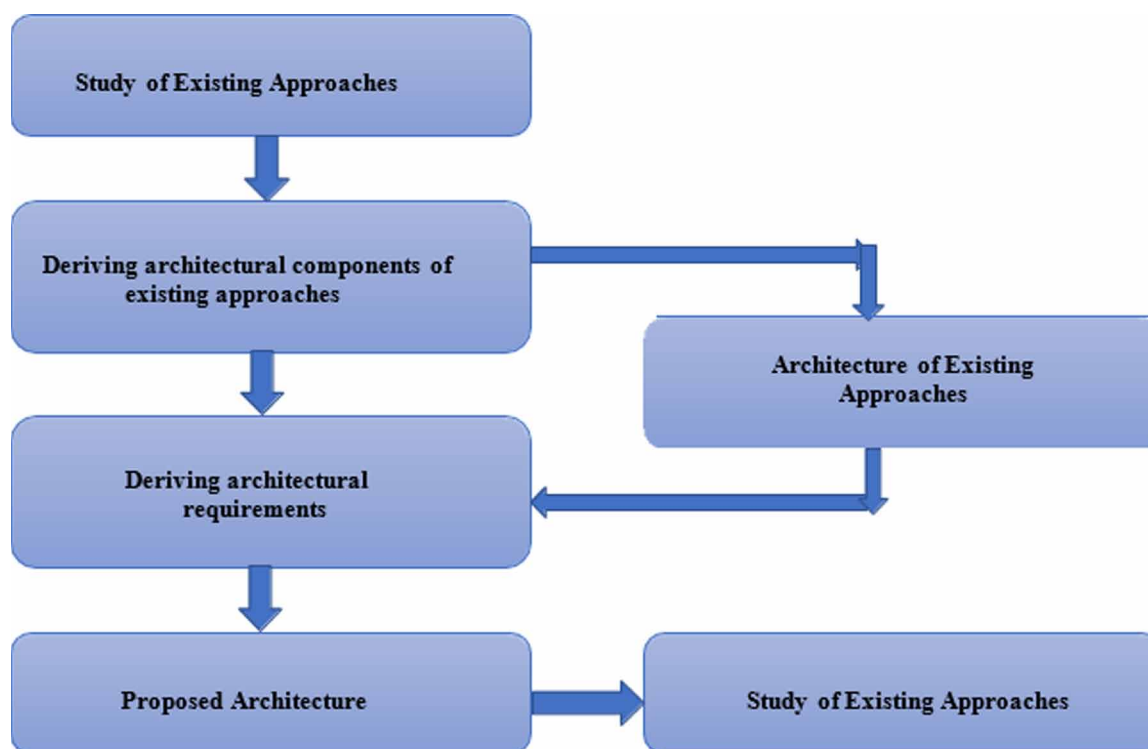
A packaged app is a zip file that contains all the resources and it enables a Browser Operating System app to function, with an app explicit in the zip's primary directory. The app explicit gives niceties about the app like as its description, icons are used to recognize the installed app and like this. The package is then used to install the app to Firefox Operating System devices. Once installed the app runs on the device but it is still able to access resources on the Web, like as a database on a web server.

The principle aim of this dissertation is to handle the illegal OAuth flows in a packaged web application system to secure the OAuth protocol. Considering the major security problems of OAuth flow, we designed and develop a packaged web application with a Web Runtime feature that gives more security facilities than other existing approaches. We developed the packaged web application with the help of app packaging which is a secure app development process. As the app package contains web resources like HTML files, JavaScript files, CSS files, media assets (icon image, audio files, etc.) along with an installation descriptor file (or referred to as configuration document). The JavaScript files inside these packages are special, in the sense; those can make calls to standardized web APIs and to custom JavaScript APIs (Charlie, C., Ben, L., Benjamin, G. Z. & Christian, S., 2011) available on the underlying platform. These custom APIs provide access to various platform services from web content enabling these kinds of web apps to implement various use cases on that platform, like the native apps running on it. We created a special web framework or runtime, W, of the platform manages the lifecycle of such packaged web apps. Here managing lifecycle refers to the cycle of installation, instantiation, and uninstallation of an app. When a packaged web app is instantiated, W creates a web view: a window where the web contents of the app are rendered on to using a web rendering engine. Within the context of the web view the app invokes various standard and custom JavaScript APIs. The runtime framework W also takes care of ensuring secure access to platform services using JavaScript API calls. During execution W may create more than one web view as per the JavaScript API call from the app code. This approach will give both the security and better performance factor.

The problem can be stated in short as from existing approaches of OAuth protocol, we selected better twos to compare with our new developed architecture OAuth for packaged web application with respect to different security and performance scenarios. The steps of the methodology are depicted as follows:

Each OAuth based systems have their own mechanism for OAuth flow. The existing approaches have their own architecture, OAuth flow, security mechanism, etc. So, we found the similarities and dissimilarities of those existing approaches to find out the problems and vulnerabilities.

To find out the architectural requirements of an OAuth based system for a client-side web application, we have analyzed the architectures of two existing systems. We gathered some useful information by studying and browsing the internet about those approaches. For more information, we tested them on the internet and analyzed the various sections of those systems. From all that information, we have tried to draw the general architectures of those approaches mentioned above. As, in this research our task is to propose, design and develop a new OAuth architecture, we must derive some architectural requirements.

OAuth 2.0*Figure 1. Steps of methodology*

Since OAuth is an authentication protocol, there is plenty of research avenues and challenges that can be explored. These research areas are security, social-technical impacts, performance, etc. (Jason, B., Elie, B., Divij G. & John, C. M., 2010). However, this research will focus on the challenges facing security of user data stored in resource servers and transmit through the internet. In this paper we see the new OAuth security architecture for packaged web apps in comparison with other two approaches. If the flows of OAuth 2.0 are similar in user's visibility, so attackers might set `response_type = token`, and obtain the access token from the browser by a script except user's awareness. The receiver, excepting an attacker, of the code or access token might log in to RP, such as the victim, and obtain the victim's account. If the State parameter is spoiled, another countermeasure in opposition to CSRF attack (Yacin, N., Prateek, S. & Dawn, S., 2009) must be taken specified in the protocol. In practice, CSRF attack is mostly disregarded through developers. If obtaining Authorization Code and using it earlier on victim, the attacker might embody the victim and log in to RP, and that has a bigger risk if the code might be used for many RPs, e.g. the code sent to RP1 might be used to log in to RP2. Due to the DE cryptographic (William, K., Robertson, Giovanni V., 2009) design of OAuth 2.0, the access token is mostly performing as a bearer token (Jones & Hardt., 2012). That is to say, any bearer of the token, e.g. an attacker, might access or manage user's profile (Prateek, S., David M. & Livshits, B., 2011).

OAuth 2.0

OVERVIEW OF OAuth 2.0 IN PACKAGED WEB APP

In the tradition client-server authentication design, the client requests an access-limited resource (protected source) on the server through authenticating with the server by using the resource owner credentials. So that, provide 3rd party applications access to limited resources, the resource owner shares its identity with the 3rd party (Mike, T. L. & Venkat, V., 2009). This creates different limitations and problems:

- 3rd party applications are necessary for storing the resource owner credentials for oncoming use, generally a password in clear-text.
- The servers are necessary to client password authentication, notwithstanding of security vulnerabilities inherent in passwords.
- 3rd party applications acquire too broad access to the resource owner preserved resources exploit resource owners except any ability to limit duration or access to a restricted subset of resources.
- The resource owners may not call access to an individual 3rd party except calling access to all 3rd parties and must do therefore by changing the 3rd party's password. Compromise of any 3rd party application outcomes in trade-off the end user's password & all of the data out of danger through that password.

Instead of utilizing the resource owner's credentials to access restricted resources, the client gains an access token a string denoting a lifetime, scope, and alternative access properties. Access tokens are issued to 3rd party users through the authorization server with the grant of the resource owner. The user uses the access token to get the restricted resources reserved within the resource server.

Roles

OAuth defines 4 prefaces:

Resource owner: An entity equipped for giving access to a reserved resource. once the resource owner may be a person, it's spoken as an end-user.

Resource server: The server facilitating the preserved resources, worthy of acceptive & reacting to preserved resource requests utilizing access tokens.

Client: The application creating preserved resource requests in favor of the resource owner and with its approval. The term "client" does not indicate any specific implementation properties.

Authorization server: The server giving access tokens to the user afterward effectively authenticating the resource owner & gaining authorization.

The interaction within the resource server & authorization server is on the far side of the Opportunity of this specification. The authorization server might be the very same server as much the resource server or a different entity (Adam, B., Juan, C. & Dawn S., 2009). An individual authorization server might issue access tokens adopted by many resource servers.

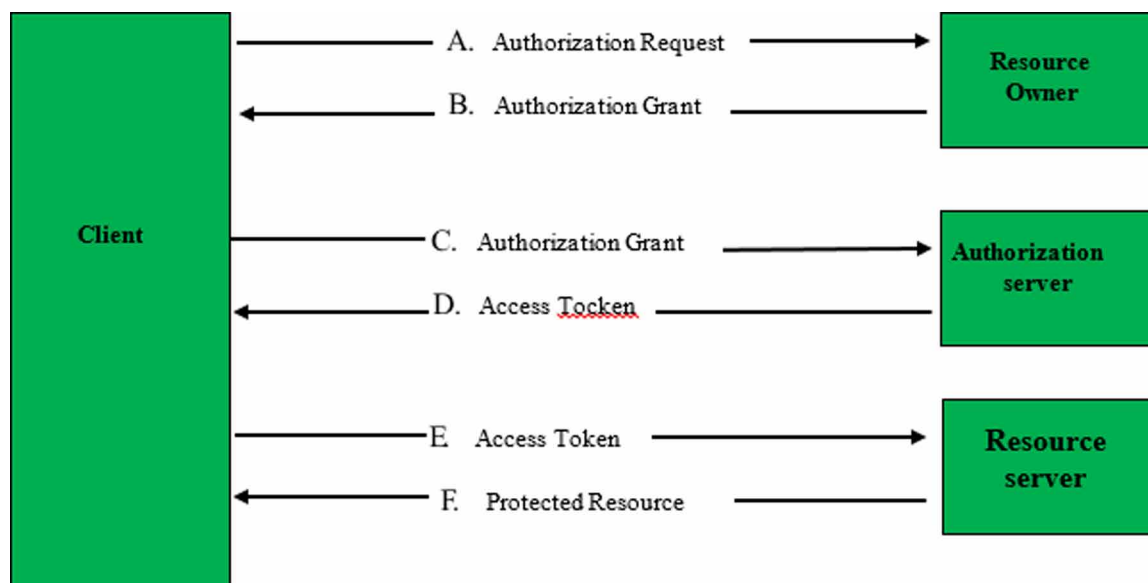
OAuth 2.0**Protocol Flow**

The summary of OAuth 2.0 flow (Hardt Ed., 2012) painted in Figure 2 recounts the interaction among the four roles and included bellow steps:

- A. The user requests authorization from the resource owner. The authorization request maybe is also created on direct to the resource owner (as shown), or ideally indirectly by the authorization server as a mediate. The user requests authorization from the resource owner. The authorization request may be is created on direct to the resource owner (as shown), or ideally indirectly by the authorization server as a mediate.
- B. The user gets an authorization to allow, that is a credential representing the resource owner's authorization, published utilizing one in every of four grant types outlined during this specification or utilizing an expansion allow type. The authorization allow type to rely on the tactic employed by the user to request authorization & also the types Through authorization server.
- C. The user asks for an access token through validating with the authorization server & exhibiting the authorization grant.
- D. The authorization server confirms the user and approves the authorization grant, and if legitimate, issues an access token.
- E. The user asks for the preserved resource from the resource server and confirms through exhibiting the access token.
- F. The resource server approves the access token, and if legitimate, serves the demand.

The favored strategy for the user to gain an authorization grant from the resource owner (portrayed in steps (A) and (B)) is to utilize the authorization server as a mediate.

Figure 2. Abstract Protocol Flow of OAuth 2.0



OAuth 2.0

Authorization Grant

An authorization grant may be a credential representing the resource owner's approval (to get its preserved resources) utilized through the user to gain an access token (Richer, Mills & Tschofenig., 2012). This detail characterizes four grant varieties –

- Authorization code
- Implicit
- Resource owner password credentials and
- Client credentials.

Authorization Code

The authorization code is gained by utilizing an authorization server as a mediate within the resource owner & the user. Rather than asking for approval straightforwardly from the resource owner, the user guides to the resource owner to an authorization server that so coordinates the resource owner back to the user with the authorization code.

Implicit

The implicit grant is a rearranged authorization code stream upgraded for users implemented in a browser utilizing a scripting language for example JavaScript. In the implicit flow, rather than issuing the user authorization code, the user is issued an access token specifically (as the consequence of the resource owner authorization). The grant type is implicit, as no halfway credentials (for example an authorization code) are issued (and later used to gain an access token).

Resource Owner Password Credentials

The resource owner password credentials (i.e. password and username) may be utilized directly as an authorization grant to gain an access token. The credentials should just be utilized when there is a high level of trust within the user & the resource owner (e.g. the client is a piece of the device operating system or a profoundly advantaged application), and when several authorizations grant types are not accessible (like an authorization code).

Client Credentials

The user credentials (or other forms of user authentication) may be utilized as an authorization grant once the authorization scope is restricted to the preserver resources under the control of the user, or to preserved resources antecedently organized with the authorization server.

Access Token

Access tokens are credentials accustomed to accessing preserved resources. An access token could be a string speaking to an authorization issued to the user. The string is sometimes opaque to the user. Tokens

OAuth 2.0

represent specific Opportunities and span of access, granted by the resource owner, and executed by the resource server & authorization server. The access token gives an abstraction layer, substitution totally different authorization constructs (for example username and password) with one token understood through the resource server.

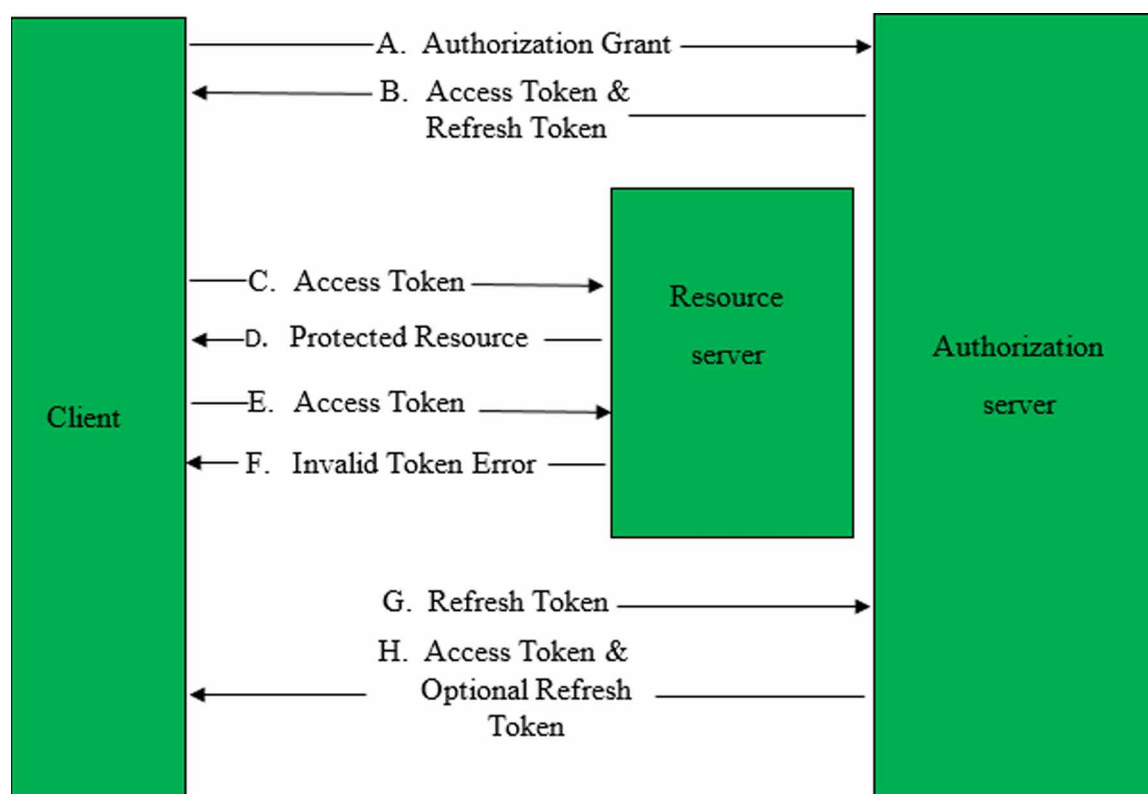
Refresh Token

Refresh tokens are credentials accustomed get access tokens. Refresh tokens (San-Tsai S., Kirstie, H. & Konstantin, B., 2012) are issued to the user through the authorization server and are accustomed gain a new access token whereas the current access token becomes void or expires, or to get extra access tokens with uniform or narrower Opportunity. Issuing a refresh token is volitional at the prudence of the authorization server. On the off chance that the authorization server issues a refresh token, it is comprised whereas issuing an access token.

The stream showed in Figure 3 incorporates the accompanying advances:

- A. The client application asks for an access token through authenticating with the authorization server & introducing an authorization grant.
- B. The authorization server verifies the client application & approves the authorization grant, and if legitimate, issues an access token & a refresh token.

Figure 3. Refreshing an expired access token



OAuth 2.0

- C. The client application makes a preserved resource demand to the resource server by showing the access token.
- D. The resource server approves the access token, and if legitimate, serves the demand.
- E. Steps (C) and (D) reiteration until the access token expired. On the off chance that the client application knows the access token lapsed, it skips to step (G); else, it makes another secured resource ask.
- F. Since the access token is void, the resource server restores a void token error.
- G. The client application asks for another access token through authenticating with the authorization server & introducing the refresh token. The client application authentication necessities depend on the user type & on the authorization server arrangements.
- H. The authorization server authenticates the client application & verifies the refresh token, and if legitimate, issues another access token (and, alternatively, another refresh token).

Interoperability

OAuth 2.0 furnishes a rich authorization framework with very much characterized security properties. In any case, as a rich and profoundly extensible framework with numerous discretionary parts, all alone, this particular is probably going to deliver an extensive variety of non-interoperable executions.

OAuth Endpoints

OAuth 2.0 utilizes two endpoints, The Authorization endpoint, and the Token endpoint.

Authorization Endpoint: The Authorization endpoint is utilized to collaborate with the resource owner and obtain the authorization to access the preserved resource. To begin with, the service will divert you to Google in order to authenticate (in the event that you are not as of now signed in) and afterward you will get an assent screen, where you will be requested to approve the service to access some of your information, e.g., your email address and your list of contacts.

Token Endpoint: The Token endpoint is utilized through the application to gain an Access Token or a Refresh Token. It is utilized through all grant types, outside of Implicit grant (whereas an Access Token is issued straightforwardly). In the Authorization Code grant, the application alternates the authorization code it obtained from Authorization endpoint for an Access Token (Hardt Ed., 2012).

SECURITY ISSUE ON OAuth 2.0 FOR PACKAGED WEB APP

As a versatile and extensible framework, OAuth's 2.0 security concerns rely upon several factors. The subsequent sections give implementers with security rules concentrated on the three user profiles, native application, and user-agent-based apps. An extensive OAuth 2.0 security model and analysis, likewise as background for the protocol diagram, is given through the OAuth-Threat model (Nazmul, H., Md., A. H., Md., Z. H., Md., H. I. S. & Shawon, R., 2018).

OAuth 2.0**Client Authentication**

The authorization server builds up user credentials with web application user with the end goal of user authentication. The authorization server is motivated to consider potential user authentication implies than a user password. Web application users must guarantee the privacy of user passwords and other client credentials. The authorization server should not issue user passwords or different user credentials to native applications (e.g. Facebook, Google, etc.) (Kitura., 2018) Or user-agent-based application users for the motive of user authentication. The authorization server can issue a user password or different credentials for the installation of a native application user on a particular device.

Client Impersonation

A malevolent user may impersonate another user and gain access to preserved resources if the impersonated user fails to, or is unable to, keep its user credentials confidential (Suresh, C., Charanjit, J. & Arnab R., 2011). The authorization server should authenticate the user whenever possible. In the event that the authorization server can't authenticate the user because of the user's nature, the authorization server should need the registration of any redirection URI utilized for receiving authorization responses and must use different intends to shield resource owners from like potentially malicious users.

Access Tokens

Access token credentials (and in addition any private access token properties) must be kept secret in transit & storage, and just apportioned among the resource servers, the authorization server, the access token is legitimate for, and the user the access token be issued to whom. Access token credentials should just transmit to utilize TLS (Wanpeng, L. & Chris J. M., 2014) with server authentication. When utilizing the Intrinsically grant type, the access token is sent in the URI section, which may open it to unapproved parties.

Refresh Tokens

Authorization servers can issue revitalize tokens to the web application users and local application users. Refresh tokens definitely kept privet in transit & storage and shared just only the authorization server and the user the refresh tokens were issued to whom. The authorization server must sustain the binding within a refresh token & the user to whom it was issued. Refresh tokens should just be transmitted utilizing TLS with server authentication.

Authorization Codes

The transmission of authorization codes ought to be made over a safe channel, and the user ought to require the utilization of TLS with its redirection URI if the URI recognizes a network resource. Since authorization codes are sent by means of user-agent redirections, they might possibly be revealed by user-agent history & HTTP referrer headers (Rui, W., Shuo, C. & Xiaofeng, W., 2012).

OAuth 2.0

Authorization codes work as plaintext carrier credentials used to confirm that the resource owner who allowed authorization at the authorization server is a similar resource owner coming back to the user to complete the procedure.

Authorization Code Redirection URI Manipulation

When asking authorization utilizing the authorization code grant type, the user may specify a redirection URI by means of the “redirect_uri” parameter (Lodderstedt, McGloin & Hunt., 2013). In the event that an attacker may control the value of the redirection URI, it may be because of the authorization server to divert the resource owner client operator to a URI under the control of the attacker with the authorization code. An attacker may make an account at a valid user and start the authorization flow. At the point when the attacker’s user-agent is transmitted to the authorization server to allow access, the attacker possession the authorization URI provided through the valid user and replaces the user’s diverted URI with a URI under the control of the attacker. The attacker at that point traps the victim into following the manipulated link to approve access to the valid user.

Resource Owner Password Credentials

The resource owner password credentials allow type is frequently utilized for legacy or migration reasons. It lessens the overall risk of storing passwords and usernames through the user but doesn’t wipe out the need to uncover exceptionally favored credentials to the user. This grant type conveys higher jeopardy than different grant types as a result of it maintains the password anti-pattern this protocol seeks to get off. The user might misuse the password, or the password might surreptitiously be published to an attacker (for example, by means of log files or different records kept through the user).

Request Confidentiality

Refresh tokens, Access tokens, Resource owner passwords, and user credentials certainly not be sent in the clear. Authorization codes would not be sent in the clear. The “scope” and “state” parameters shouldn’t include sensitive user or resource owner data in plain text, as they may be sent over shaky channels or stored shakily.

Ensuring Endpoint Authenticity

In order to obstruct man-in-the-middle attacks, the authorization server certainly needs the utilizes of TLS (Transport Layer Security) with server authentication for any asks transmitted to the authorization and token endpoints. The user certainly approves the authorization servers.

Credentials-Guessing Attacks

The authorization server certainly obstructs attackers from guessing refresh tokens, access tokens, resource owner passwords, authorization codes and client credentials (San-Tsai, S. & Konstantin, B., 2012). The possibility of an attacker guessing produced tokens (and different credentials not intentional for handling

OAuth 2.0

through end-users) certainly be less than or equal to 2-128 and could be less than or equal to 2-160. The authorization server certainly uses different means to defend credentials desired for end-user usage.

Phishing Attacks

Wide deployment of this and analogous protocols can cause end-users to become satisfied with the practice of being redirected to websites where they are inquiring to enter their passwords. If end-users aren't cautious to validate the authenticity of these websites earlier on entering their credentials, it will be feasible for attackers to utilize this practice to snatch resource owners' passwords. Service providers would try to educate end-users about the risks phishing attacks posing and would give mechanisms that create it simple for end-users to ensure the authenticity of their sites (Daniel, F., Ralf, K. & Guido, S., 2016).

Cross-Site Request Forgery

Cross-site request forgery (CSRF) (Caimei, W., Yan, X., Wenchao, H., Huihua X., Jianmeng, H. & Cheng S., 2017) is an adventure in which an attacker because the user- agent of a prey end-user to take after a malevolent URI (for example, gave to the user-agent as a deceptive link, redirection or image) to a confiding in server (generally established by means of the presence of a substantial session cookie. Once authorization has been gained from the end-user, the authorization server diverts the end- user's user-agent back to the user with the needed binding value included in the "state" parameter. The binding value enables the user to confirm the validity of the request by coordinating the binding value to the user agent's confirmed state. The binding value utilized for CSRF protection certainly holds a non-guessable value, and the user agent's confirmed state (for example, HTML5 local storage, session cookie) certainly be kept in a place accessible just once to the user and the user-agent (i.e., preserved through the same-origin policy). A CSRF attack in opposition to the authorization server's authorization endpoint may outcome in an attacker gaining end-user authorization for a bitchy user except alerting or involving the end-user. The authorization server certainly developed CSRF protection for its authorization endpoint and confirm that a malicious user cannot gain authorization except the awareness and evident consent of the resource owner.

Clickjacking

In a clickjacking attack, an attacker records a valid user and then built a malicious site in which it stacks the authorization server's authorization endpoint web page in a diaphanous iframe overlaid over a set of artificial buttons, that's are gingerly build to be placed specifically under significant buttons on the authorization page. When an end-user clicks a deceptive seen button, the end-user is really clicking an unseen button on the authorization page (e.g. an "Authorize" button). This permits an attacker to ruse a resource owner into allowing its user access except for the end-users knowledge.

Code Injection and Input Validation

A code injection attack happens when an input or in a different way the exotic variable is utilized through an application unsanitized and because of shuffling to the application logic. This can grant an attacker to obtain access to the application device or its information, because of denial of service, or introduce a

OAuth 2.0

spacious range of malicious side-effects. The authorization server and user certainly sanitize (and affirm when possible) any value accepted in individual, the value of the “redirect_uri” and “state” parameters.

Open Redirectors

The authorization server, client redirection endpoint, and authorization endpoint can be wrongfully configured and handle as free redirectors (OpenID Authentication 2.0 - Final., 2007). An open redirector is an endpoint conducting a parameter to automatically redirect a customer-viceregent to the position specified by the parameter value except any affirmative. Open redirectors can be applied in phishing invasions, or by a charger to catch end-users to visit malicious sites by applying the URI authority material of a homely and believable target (Victoria, B., 2016).

Misuse of Access Token to Impersonate Resource Owner in Implicit Flow

For public clients using implicit streams, this specification does not give anyway for the client to determine what client an access token was issued to. A resource proprietor may agreeably surrogate entree to a resource by offering an access token to an attacker’s bitchy client. This may be payable to phishing or some other plea. An attacker may also snatch a token through several mechanisms. An aggressor may then try to incarnate the resource proprietor by giving the access token to a legal public client. Servers connecting with local applications that count on being adopted an access token in the backchannel to define the customer of the client may be likewise compromised by an attacker making a compromised application that can impel unrestricted robbed access tokens. (Francisco, C. & Karen, P. L., 2011)

EXISTING APPROACHES

OAuth is an uncovered security measure that enables customers to give appointed and time cramped rights to an application to entrance preserved customer’s resources, gathered on several exotic resource servers, except needing them to divide their credentials, with this application. Handling OAuth, a client application has one ingress token for more use with an HTTP redirect reaction from the resource server sometimes the user authenticates the resource entry (lhshaoren., 2010). OAuth 2.0 stream is, particularly for user authorization. It is planned for applications that can hoard private news and sustain the realm. A rightly authorized web server application can entrance an API while the customers communicate with the application or afterward the user has left the application.

Approach A: OAuth 2.0 to Access Google APIs

The approach A explained in (Google Identity Platform, 2018) suggests specifying the redirect URL as a loopback IP address like `http://127.0.0.1::port` or `http://::1::port` and requires the client app to start listening to a randomly available HTTP port. To receive the access token using a loopback IP address, the app must be listening to the local webserver.

OAuth 2.0**The Basic Pattern of OAuth 2.0 to Access Google APIs**

To initiate, get OAuth 2.0 client certificates from the Google API Console. Then the client application entreats an ingress token from the Google Authorization Server, gets a token from the reaction and transmits the token to the Google API that the user wants to access. Every application chases a fundamental pattern when engrossing a Google API conducting OAuth 2.0. At an exalted level, follows four steps.

- **Obtain OAuth 2.0 credentials from the Google API Console:** The Google API Console to make OAuth 2.0 certificates like a client ID and client covert that are acquainted with both Google and user applications. The group of values changes founded on what form of application is constructing. For example, a JavaScript application does not claim a hidden, however, a web server application does.
- **Obtain an access token from the Google Authorization Server:** Before an application can penetration personal data using a Google API, it must gain an entrance token that gives access to that API. An individual access token can grant asymmetrical degrees of ingress to multiple APIs. A changeable parameter named scope monitors the group of resources and actions that an access token permits. During the access-token request, an application gives one or more values in the scope parameter.
- **Send the access token to an API:** After an application gain an access token, it deputizes the token to a Google API in an HTTP authorization header. It is allowable to give tokens as URI query-string parameters because URI parameters can end up in log records that are not fully secure. Access tokens are valid only for the group of activities and resources narrated in the scope of the token request.
- **Refresh the access token, if necessary:** Access tokens have restricted lifetimes. If an application wants access to a Google API behind the lifetime of an individual access token, it can favor a refresh token.

Scenarios of OAuth 2.0 to Access Google APIs

Generally, OAuth provides clients (e.g. Mobile app) a 'secure confer penetration' to server funds on the part of a resource owner. The Web-server applications scenario is applied to authenticate a web application with a third-party service. There are many scenarios of OAuth 2.0 to access Google APIs.

Web Server Applications

The Google OAuth 2.0 endpoints backings web server applications that usage languages and molds such as PHP, Java, Python, Ruby, and ASP.NET (Android WebView., 2018). The authorization series starts when an application redirects a browser to a Google URL; the URL involves query parameters that imply the type of ingress being petition. Google manages user authentication, session selection, and user consent. The outcome is an authorization code, which the application can interchange for an access token and a refresh token.

OAuth 2.0

Figure 4. Abstract Protocol Flow of Web server applications OAuth 2.0

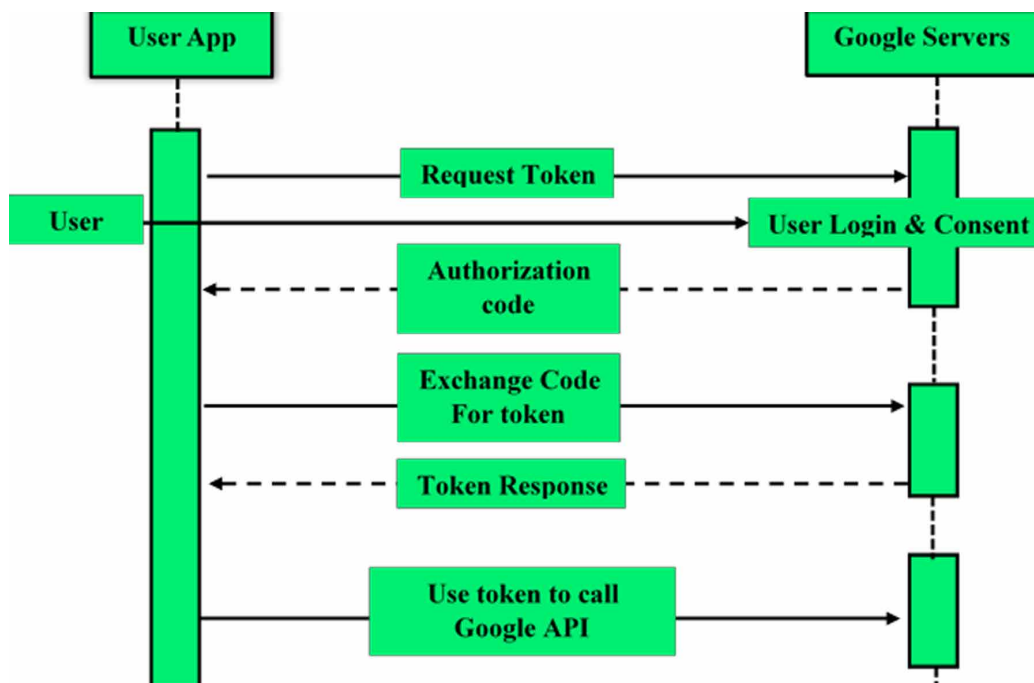
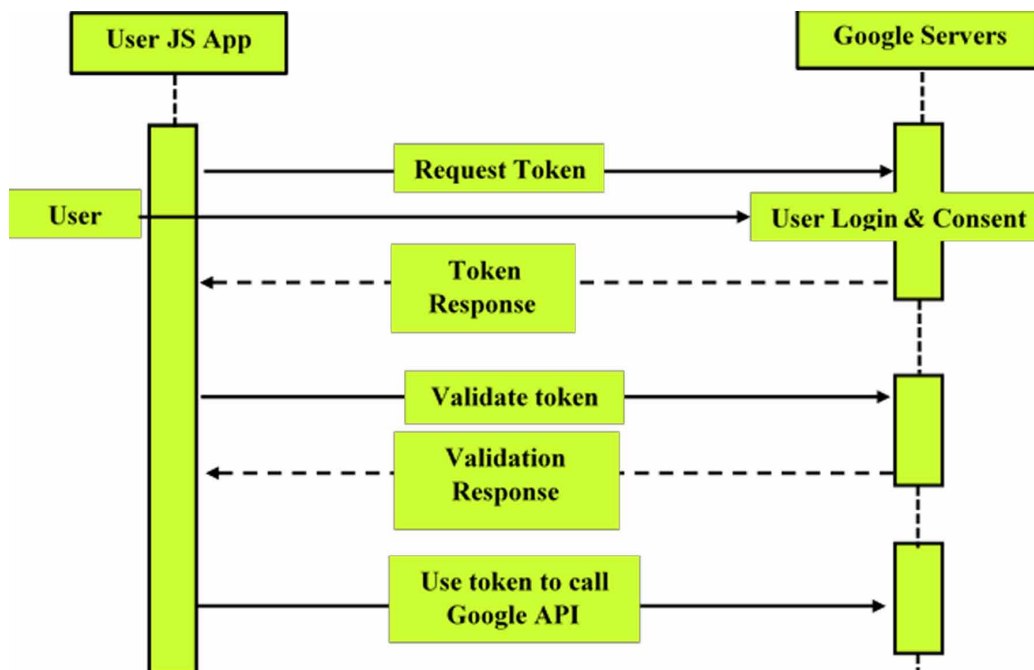


Figure 5. Abstract Protocol Flow of Installed applications OAuth 2.0



OAuth 2.0**Installed Applications**

The Google OAuth 2.0 endpoints clinch applications that are installed on devices like computers, mobile devices, and tablets. When user creates a client ID from the Google API Console, specify that this is an installed application, then choose Chrome, iOS, Android, or “Other” as the application type.

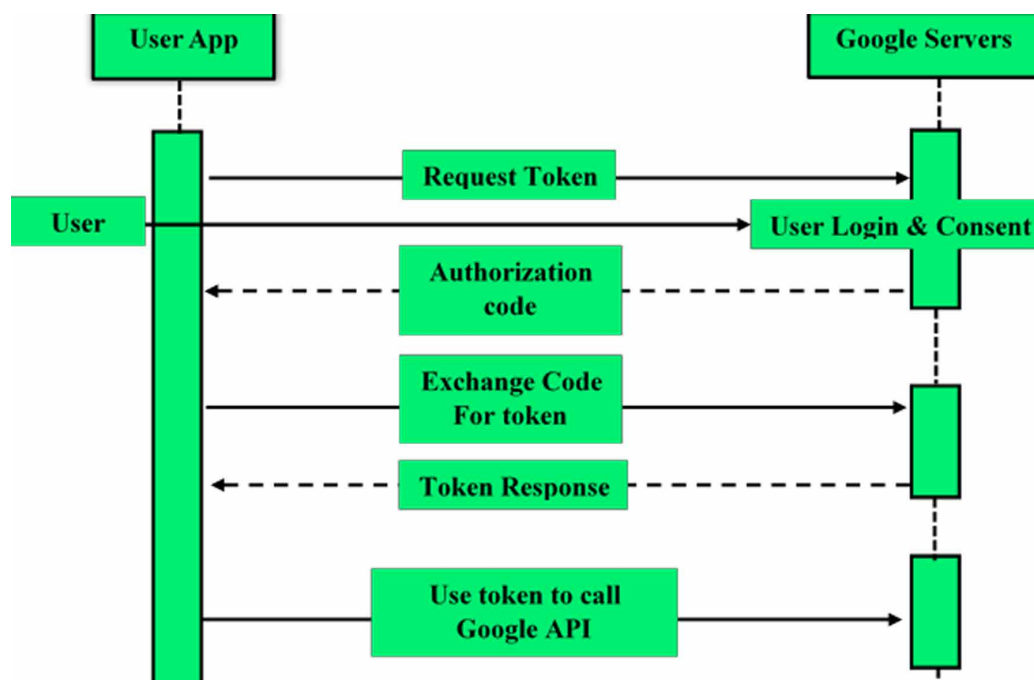
The procedure output in a client ID and, in some law, a client covert, which user embed in the source code of the user’s application. The authorization sequence begins when user’s application redirects a browser to a Google URL; the URL added query parameters that introduce the type of ingress being asked. Google maintains the session selection, user authentication, and user consent. The output is an authorization code, which the application can interchange for a refresh token and an access token. The application should hoard the refresh token for next usage and usage the access token to ingress a Google API. At one time the access token died, the application uses the refresh token to gain a new one.

Client-Side Applications

The Google OAuth 2.0 endpoint supports client-side applications that run in a browser.

The authorization sequence begins when the user’s application redirects a browser to a Google URL; the URL adds query parameters that select the sampling of entree being asked. Google manages the session selection, user consent, and user authentication. The output is an access token, which the client should verify before adding it in a Google API request. When the token departs, the application repetitions the procedure.

Figure 6. Abstract Protocol Flow of Client-side applications OAuth 2.0



OAuth 2.0

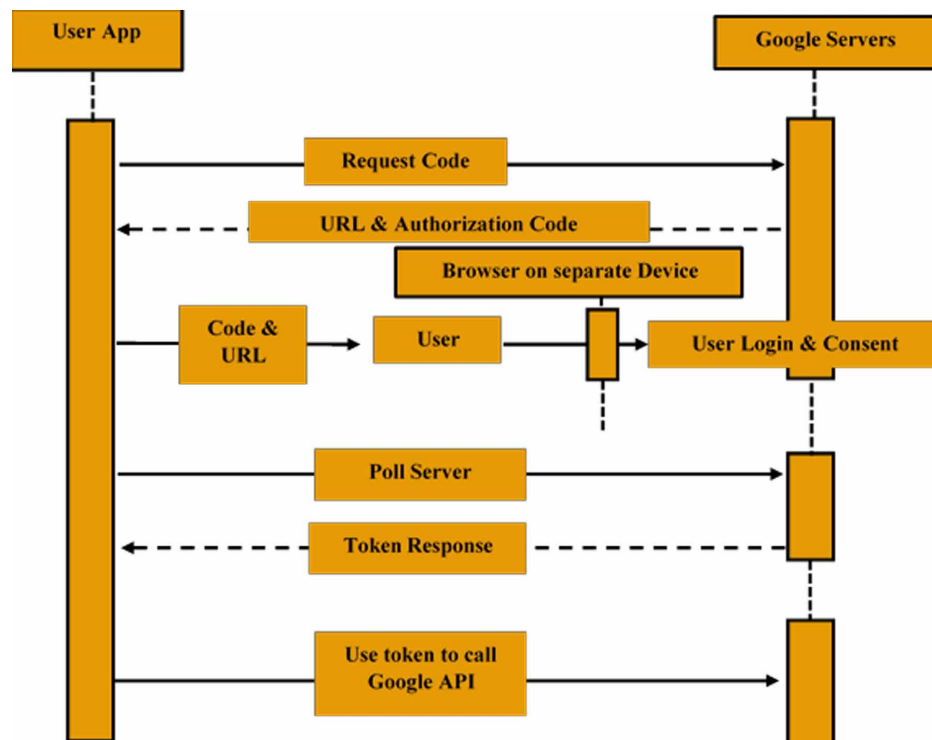
Applications on Limited-Input Devices

The Google OAuth 2.0 endpoint helps applications that move on limited-input instruments such as video cameras, game consoles, and printers. The authorization order starts with the application creating a web service that asks a Google URL for an authorization code. The reaction contains different parameters, adding a URL and a code that the application displays to the user.

The user gains the URL and code from the apparatus, then switches to individual instruments or computers with larger input capabilities. The user opening a browser navigates to the specified URL, logs in, and enters the code. Hitherwards, the application peaks a Google URL at a specified interim. After the customers approve ingress, the answer from the Google server holds an access token and a refresh token.

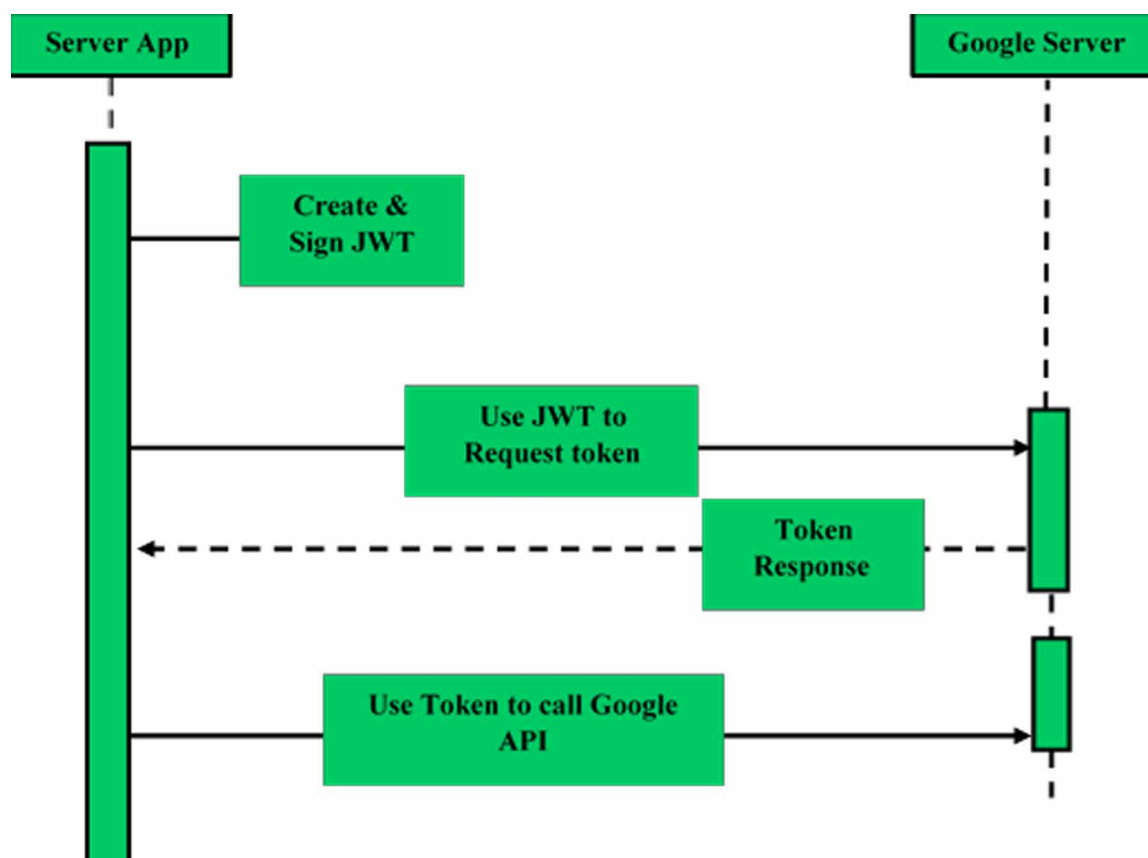
The application should gather the refresh token for the next usage and usage of the access token to ingress a Google API. At one time the access token dies, the application conducts the refresh token to gain a new one

Figure 7. Abstract Protocol Flow of limited-input devices applications OAuth 2.0



Service Accounts

A service account's credentials, which user obtains from the Google API Console, involve a propagated email address that is identical, a client ID, and at least one public/private key couple. A user uses the client ID and a private key to make a signed JWT and build access-token solicitation in the opposite

OAuth 2.0*Figure 8. Abstract Protocol Flow of Service account applications OAuth 2*

ordination. The application then gives the token indictment to the Google OAuth 2.0 Authorization Server, which retires an access token. The application demeanor the token to entrance a Google API. When the token kicks off, the application repetitions the methods.

Token Expiration

Users must write their code to relate the probability that an approved refresh token might no longer active. A refresh token might pause working for one of these reasons:

- The user has rejected the app's ingress.
- The refresh token has not been demeanor for six months.
- The user altered passwords and the refresh token take on the Gmail tract.
- The user account has surpassed the greatest number of inflicted (live) refresh tokens.

There is now a border of 50 refresh tokens per customer account per client. If the boundary is touched, making a new refresh token automatically defeats the ancient refresh token without inflaming.

OAuth 2.0

Limitations of OAuth 2.0 to Access Google AP

In this approach it receives the access token using a loopback IP address, the app must be listening to the local webserver. This approach recommends that the client app, upon receiving the token response, should show an HTML page instructing the user to close the browser tab and return to the client app. This approach also allows using localhost in place of loopback IP, if that does not lead to any firewall related issue in the underlying platform. In this approach, it is possible to successfully get the access token, but OAuth flow initiated from a packaged web app is exposed to security vulnerability due to weak redirection endpoint definition and handling.

Approach B: Facebook App in Tizen

This approach (Facebook App in Tizen, 2018) suggested for performing OAuth with Facebook on the Tizen platform, is to provide a legitimate HTTP URL as redirect endpoint and continuously poll for redirection to the given redirect URL. To complete the Facebook Authentication method user has to follow three steps.

Initiate a Redirect to an Endpoint

The app must begin a redirect to an endpoint having the according to parameters:

- `client_id`: The ID of the app, established in the app's fascia.
- `redirect_uri`: The Site URL found in the "Website with Facebook Login" chapter of the app's fascia.
- `scope`: A comma differentiated list of Permissions to ask from the customer of the app.

The endpoint will show the Facebook login window. After admitting the credentials, a dialogue will look inquiring user to grant entry to his friend list, public profile, and any extra Permissions asked by the app. If the user selects "OK" on the dialogue, the app will achieve entree to specified permissions.

Child Window

The "Child Window" idea is applied to open the Facebook login window (Erika, C., & David W., 2014). At one time the user finished the login procedure, the authentication URL will give the access code to `redirect_uri`. Instruct the authentication URL to recover the 'access code' and regain to the application by sending the child window. It starts a timer that will monitor inside timeout callback the change in the URL for the child window due to redirection to retrieve the 'access code'.

Collecting the Access Code

After collecting the access code use Facebook Graph API to generate the access token. It retrieves 'access code' and initiate an AJAX request and in the success callback of AJAX request parse the access token and close the child window.

OAuth 2.0**Limitations of Facebook App in Tizen**

This approach provides a legitimate HTTP URL as redirect endpoint and continuously poll for redirection to the given redirect URL. It is possible to successfully get the access token, but OAuth flow initiated from a packaged web app is exposed to security vulnerability due to weak redirection endpoint definition and handling.

NEW OAuth SECURITY ARCHITECTURE FOR PACKAGED WEB APPS

The Internet today provides us a collaborated environment of various types of software platforms and social networking systems (Mohamed, S. & Fadi, M., 2014) (Marian, H., Markus, H., Susanne, W. & Matthew, S., 2014) Application (henceforth referred to as “app”) developers, especially client web app developers, want to utilize this medium to access various user resources, such as photos or videos maintained in some external resource server, in order to provide a richer and connected experience through their apps (Hao, H., Vicky, S. & Wenliang, D., 2013). Nowadays OAuth 2.0 is very widely used on most websites, and there is a correspondingly rich infrastructure of identity providers (IdPs) providing identity services using OAuth 2.0. Use of OAuth 2.0 (Fadi, M. & Mohamed, S., 2016) by Facebook, Google and Microsoft have previously been studied, and issues have been identified. However, security efficient OAuth implementations are not available for locally installed packaged client web apps (Tara, W. & Kori I. Q., 2005). In this section, we propose a novel OAuth implementation for these kinds of apps. We developed a packaged web app with Web Runtime environment.

After analyzing the existing approaches of OAuth 2.0 for packaged web apps we have found some similarities and dissimilarities. To develop an architecture capable of addressing the existing architectural problems, several architectural requirements have been derived and discussed.

Packaged Web App Development

To implement our own new OAuth authentication protocol, we need to develop a Packaged Web App. A Packaged Web application (Dongwan, S. & Rodrigo, L., 2011) includes all the backup files that are wanted by the Web application. Hence, a Web application may run without network connectivity or any other external resources after finishing the installation. The packaged web application model supports a big amount of standard W3C/HTML5 characteristic, which involves many JavaScript APIs besides extra HTML markups and CSS features. Those features, besides the Device APIs and UI framework support, may be used to make a rich Web application in a diversity of categories, such as device information access, multimedia, graphics, contacts, messaging, and games. For packaged web apps the installation descriptor is a very important component of the package. It declares various properties of the app that should be manifested in the system during its lifecycle. It includes the icon and name to be shown when the app is installed.

Web Runtime Environment

The Web Runtime performs a body of the core behavior of any programming language and allows it to be changed via an API or inlaid domain-specific language. The web runtime (Web Runtime – Tizen

OAuth 2.0

developer., 2018) is like it uses web-based programming languages like Java-script which utilizes the core behavior a computer language. Another example of a Web Runtime is JsLibs which is a stand able JavaScript development runtime environment for using JavaScript as a common all-round scripting language. JavaScript is used to make responsive interfaces that enhance the user experience and give dynamic functionality except having to wait for the server to respond and direct to another page. It supports:

- Management of packages (installation, update, etc.)
- Lifecycle and Execution (launching, pause, resume, etc.)
- Runtime security (API/network access, sandboxing, etc.)
- Platform and device integration (access local device and platform resources)

OAuth 2.0 Authentication Flows

The OAuth-SSO systems (Fadi, M. & Mohamed, S., 2016) are based on browser redirection that an RP redirects the Clint browser to an IP which collaborates with the Clint before redirecting the Clint back to the RP website. The IP authenticates the Clint, recognizes the RP to the Clint, and asking for consent to offer the RP access to the services and resources on the side of Clint. Once the implored consents are granted, the Clint is redirected back to the RP with an access token which illustrates the granted permissions.

With the approved access token, the RP then invokes web APIs revealed by the IP to access the client's profile properties. The OAuth 2.0 specification which defines two flows for RPs to gain access tokens: server-flow (called as the Authorization Code Grant), designed for web applications which gain the access tokens from their server-side program logic; and client-flow (called as the \Implicit Grant") for JS applications which runs in the web browser.

Authentication & Resource Server

An authentication server offers a network service that applications use to certify the credentials, typically account names and passwords, of their users. Major authentication algorithms embrace passwords, Kerberos, and public-key cryptography.

The Resource Server could be a huge collection of libraries and the web application engaged towards a lot of economical inclusion of static resources like CSS and JavaScript in Java web applications. The tip goal of those utilities is to scale back the number of distinct resource URLs the browser should recover the content from for any given page and to require advantage of browser facet caching to scale back the number of requests the browser should build.

Access Tokens

Access tokens area unit credentials used to access safe resources. associate access token could be a string describing an authorization issued to the client. The string is sometimes obscure to the client. Tokens represent special scopes and durations of access, approved by the resource owner, and insisted by the resource server and authorization server. Then the token could denote an identifier that will not retrieve the authorization info or could self-contain the authorization info in an exceedingly verifiable manner

OAuth 2.0

(i.e., a token string consisting of some information and a signature). further authentication credentials, that area unit on the far side the scope of this specification, is also needed for the client to use a token.

Fulfilling Architectural Requirements

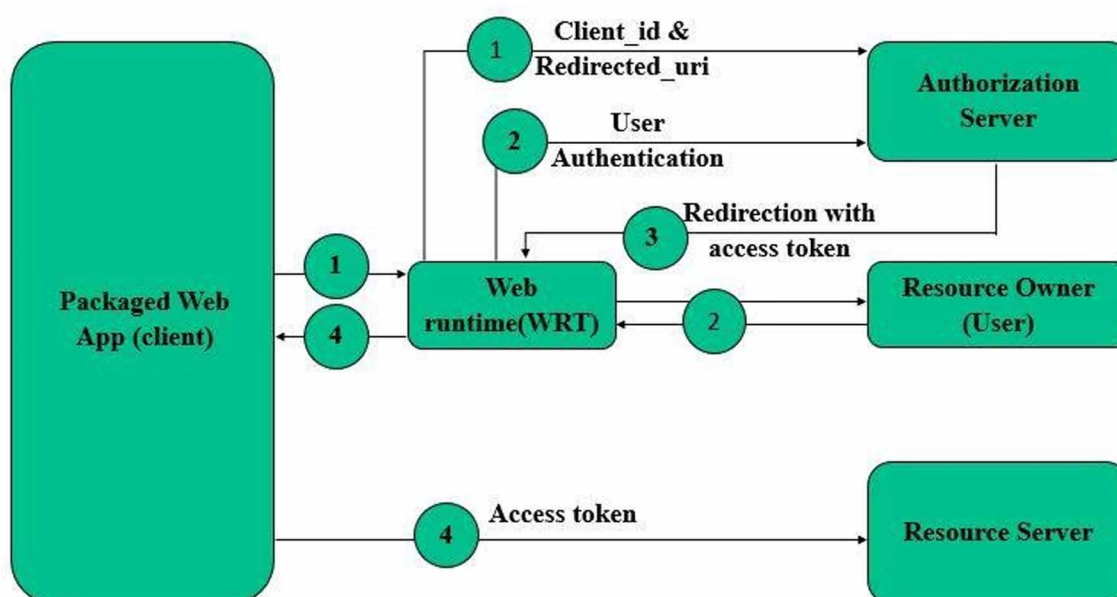
The proposed framework fulfills the requirements derives in the previous section as follows:

- The architecture ensures that access token if delivered is delivered to the right, intended, a legitimate target. The feature Web Runtime ensures that.
- The architecture makes our approach handle OAuth response faster than another two existing approaches.
- The proposed architecture performs better than the other two approaches.

PROPOSED ARCHITECTURE**Description of Proposed Architecture****Securing the OAuth Flow for Packaged Web Application**

In our proposed system, web runtime (WRT), W (Web View), is the key component that ensures a secure environment for OAuth flow execution from a packaged web app. In our system, we have considered packaged web apps because those are more aligned to W3C standards. With packaged web apps, as mentioned already, each app mentions its lifecycle properties inside installation descriptor or configu-

Figure 9. Architecture of new OAuth flow with WRT



OAuth 2.0

ration document, config.xml, by using different XML elements. In some of these XML elements are taken directly from W3C packaging specification (Dongwan, S., Huiping, Y. & Une, R., 2013) with its custom XML elements. Each app that wants to execute OAuth flow should specify special OAuth related information in its config.xml file (Kaushik, D., Prabhavathi, P. & Joy, B., 2017).

Custom XML Element OAuth

For the previously defined purpose, we have introduced a special custom XML element <OAuth> for the config.xml file (Custom Elements- W3C Working Group Note., 2018). Listing 1 depicts the part of the XSD showing this custom element. This custom element, if present, must include its mandatory redir-url attribute. The client apps need to specify in config.xml the set of redirection URLs that are already registered with the authorization server using one or more <OAuth> elements by specifying each redirected URL as the value for redir-url attribute. If there be multiple redirection URLs used by the client app, equal number of <OAuth> elements should be added to the config.xml file. The order in which the elements are specified in the config.xml file is not important. Each value of the redirection URL specified as the value for redir-url attribute must be of the form http://localhost/[optional_path_components]/<app_id>, where <app_id> is the unique id that is used by W to identify an installed packaged app. This requires the path component of the redirection URL to end with <app_id>.

Listing 1: XML Element OAuth (Custom)

```

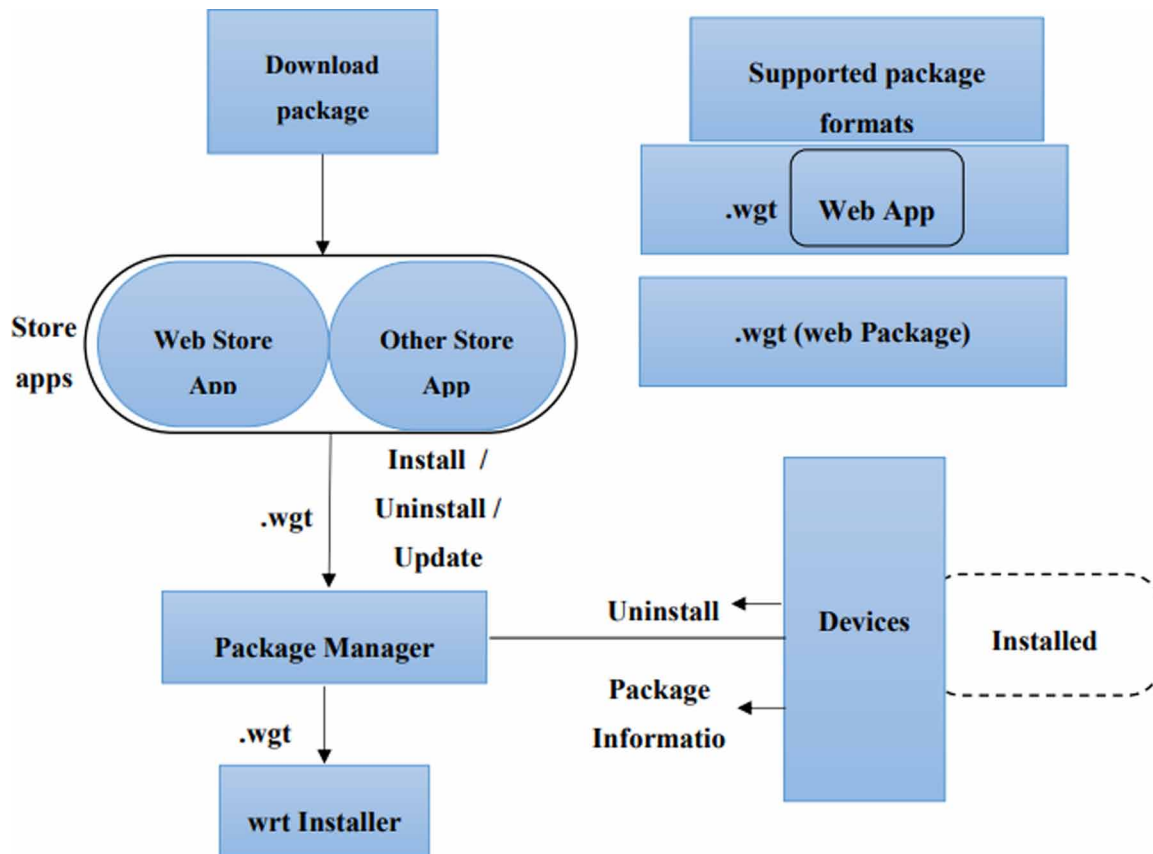
1  <xs:element name="OAuth">
2  <xs:complexType>
3  <xs:attribute name="redir-url"
4  type="xs:anyURI"
5  use="required"/>
6  </xs:complexType>
7  </xs:element>
```

The Packaging on the Web

A packaged web application includes all the support files required by the web application. Hence, a web application will run with none further resources or network property when installation. connected data the application model supports an upscale set of standard W3C/HTML5 options, that embrace varied JavaScript App is similarly as Understanding further HTML mark-ups and CSS options (McGloin & Hunt., 2011). These options, besides the Device App, is and UI framework support, can Programming be wont to produce made web applications during a type of classes, like contacts, messaging, device info access, multimedia, graphics, and games. Applications within the same package follow an equivalent installation life-cycle, handled by the application package manager.

Application Package Manager: The application package manager is one in all the core modules of our application framework, and is liable for installing, uninstalling, and change packages, and storing their data. using the package manager, we will able to additionally retrieve data associated with

Figure 10. Application package manager



Web Application Package: The software development kit packages the whole web application as a zipped file (McGloin & Hunt., 2011). Therefore, while packaged web application installation W 1st extracts the web app package to any location Lf in the file system. After that it parses the config.xml file from the desired location Lf. The web platform supports HTML, JavaScript, and CSS based Web applications and packaged following the W3C specification. The platform additionally gives device APIs to gain platform capabilities, which enables a rich Web apps development environment. An application package of Web (Hardt., 2011) must keep to the following conventions:

- The file extension and Package format

OAuth 2.0

File extension: *.wgt* (for ex. *abcde.wgt*)

File format: ZIP

MIME type: *abcde/widget*

- Application ID
After publishing the application, the App ID can't be changed.
- Contents of the Package
- File and folders: The root of the Web package the ZIP file and it contains numerous files and folders, some of these are reserved. The contents of a package show in table 1.
- Directory hierarchy (after installation on device)
- The following table 2 illustrates the Web application package directory structure.

Table 1. Package content

Name	Type	Description
config.xml	File	Application configuration document
icon.gif	File	Application default icon
icon.ico	File	Application default icon
icon.jpg	File	Application default icon
icon.png	File	Application default icon
icon.svg	File	Application default icon
index.html	File	Application default start file
index.htm	File	Application default start file
index.svg	File	Application default start file
index.xhtml	File	Application default start file
index.xht	File	Application default start file
locals	Folder	Container for localized content

WRT Operations

We already know from the previous sections that; Web Runtime environment performs part of the core behavior of any programming language. Web runtime on the OS follows:

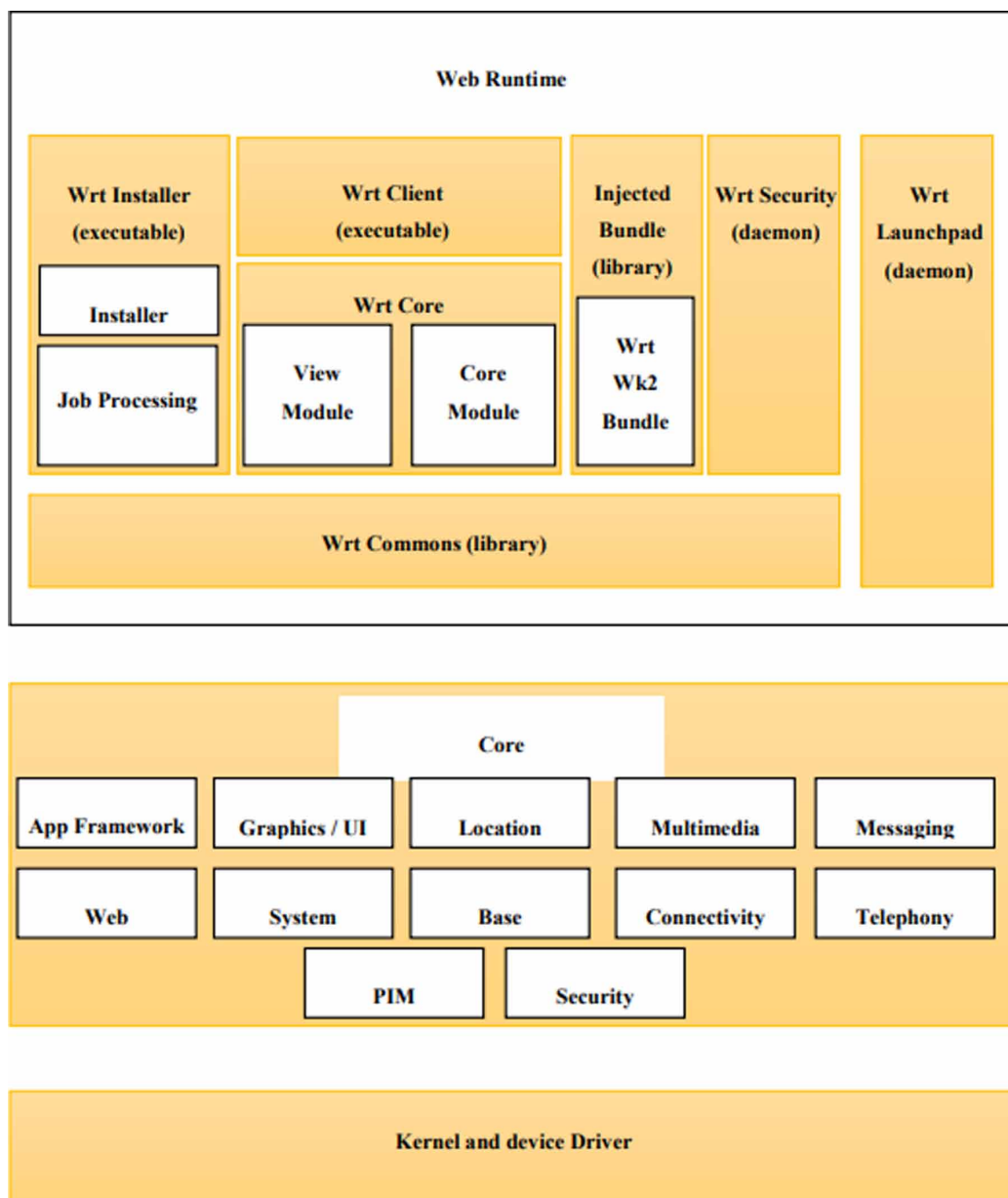
- Managing package (install, update, etc.)
- Lifecycle and Execution (launch, pause, resume, etc.)
- Security in Runtime (API/network access, etc.)
- Platform and device integration (access platform resources and local devices)

Web Runtime is the component that allows web applications to run outside the browser. The users able to install the

web apps and use web apps as a standalone application. It manages the installation, update, life cycle, system calls, API/network access, resources, platform integration, and access control of the desired web

OAuth 2.0*Table 2. Web application package structure*

Package	Root Directory	Application ID	Core XML file
app	<i>home/owner/apps_rw/<Package ID></i> For example: <i>home/owner/apps_rw/qik37po9ck</i>	<i><Package ID>.<Name></i> For example: <i>qik37po9ck.sample</i>	<i>opt/share/packages/<Package ID>.xml</i> For example: <i>opt/share/packages/qik37po9ck.xml</i>

Figure 11. Web runtime internal block

OAuth 2.0

application. Figure 11 illustrates the Internal Blocks of Web Runtime. The Web structure furnishes the best Web involvement with Browser and packaged Web Applications by means of Web Runtime. The Web Runtime is the Execution condition for packaged Web Application. It has the accompanying highlights:

- Focusing on functionality, performance, Standard Compliance(W3C)
- More gadget include availability through Tizen Device API
- jQuery Mobile based Tizen Web UI FW enables easy Web Application development

Parsing OAuth Element

While parsing the components present in the config.xml record, the parser segment of W approves every component against the specified components. The parser extracts relevant information from the element only if the element is valid and it packs with appropriate values related in-memory data structures.

Some of these data structures are also serialized securely to a database in the local file system. When a packaged web app is undergoing any lifecycle operation at some later point in time, refer to the values stored in the database for appropriate actions. Figure 12 explains these operations pictorially.

If the type of an element is *ELEMENT-OAuth* (denoting an *<OAuth>* element) the function checks if it has the mandatory *redir-url* attribute. It also checks if the value of the *redir-url* attribute follows the expected format. If yes, this function populates redirect URL member of *OAuth* element object with the value of *redir-url* attribute for further processing.

The function *PARSE-URI-FOR-APP-ID()* parses the underlying redirect URL to retrieve the app id from it. If the retrieved app id successfully matches with the app id of the client app, the redirect URL is appended to a list of redirect URLs *S* maintained for this app. This list *S* is stored permanently in a database table for this client app and is used for OAuth flow control when validating an initiated OAuth flow, as explained in the next section. This database is maintained securely so that it cannot be tampered by any means.

Each time the parser component of W encounters an *<OAuth>* element it executes the algorithm shown in Listing 2. The *ELMNT-VALID()* function checks the validity of a *config.xml* element against its type.

Listing 2: Algorithm implemented by W to parse an *<OAuth>* element encountered in config.xml

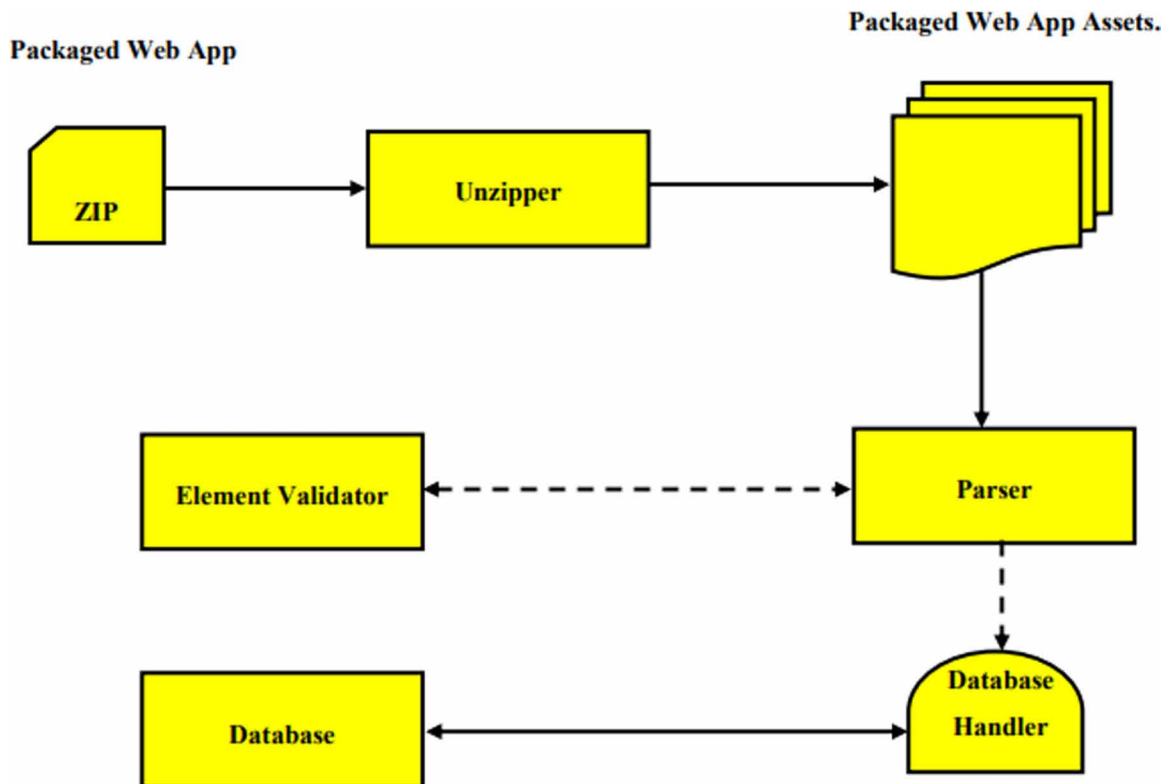
```

PARSEOAuthELMNT() // Parse OAuth element
1  w = a packaged web app; // Input
2  ε = Current OAuth element in w.configXml; // Input
3  S = Set of redirection URLs retrieved already; // Input
4  o = empty OAuth element; // local object
5  if (true == ELMNT-VALID(ε, ELMNT-OAuth, o))
6  // get app id;
7  D = PARSE-URI-FOR-APP-ID(o.redirUrl);
8  if (D == w.app_id) {
9  S = S U { o.redirUrl };
10 return TRUE;
11 else
12 return FALSE;

```

OAuth 2.0

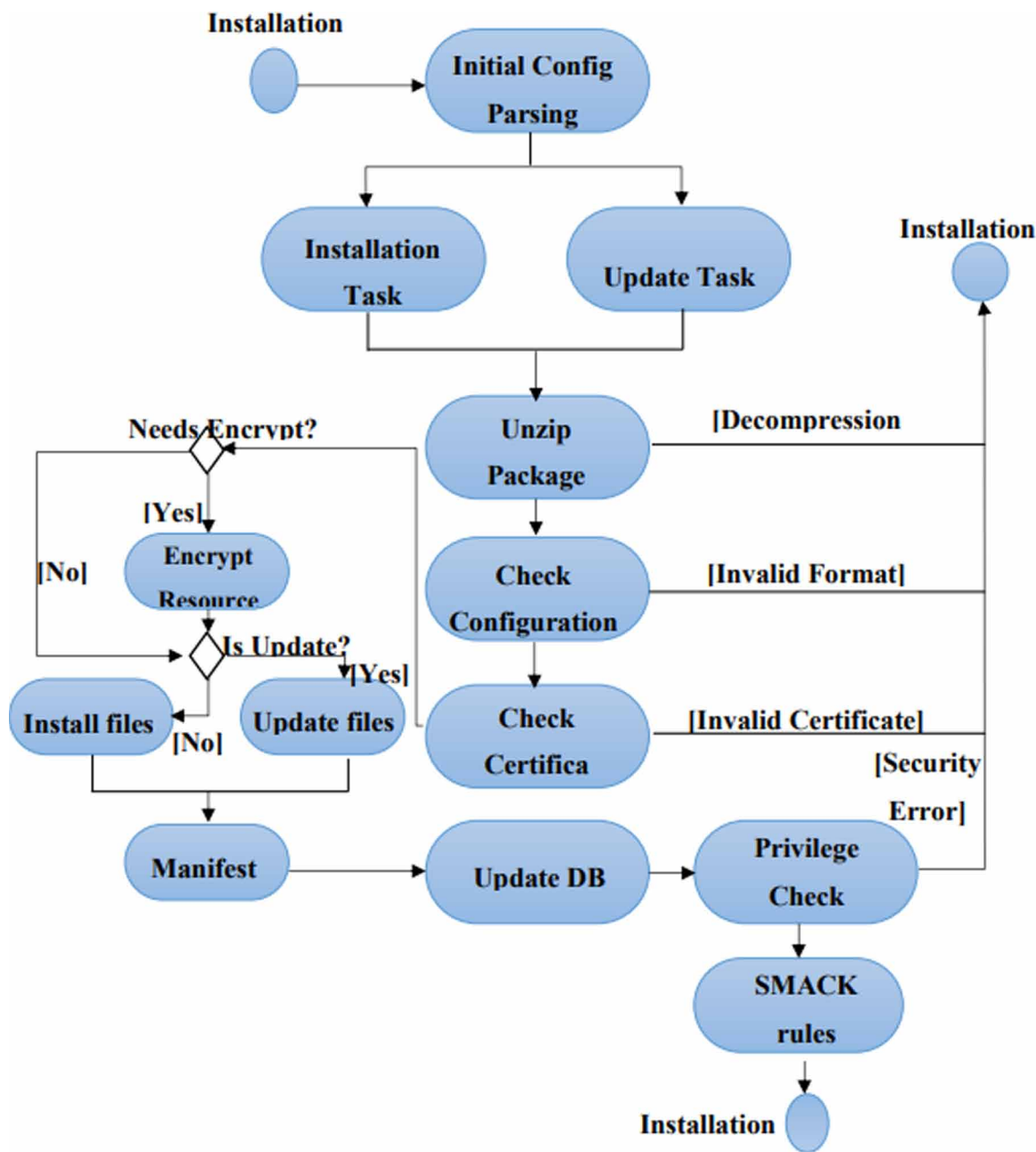
Figure 12. WRT operations during installation of packaged web application. Dotted lines indicate data flow and control flow. Solid lines indicate data flow.



Algorithm for OAuth Flow Control

In our architecture, W provides a JavaScript API to enable a packaged web app create a web view which is a mini window within the current view window and start a browsing context when it is executing. A packaged web app can create such a view only if it declares a custom privilege inside its configuration document. If an app tries to create a web view, W checks if the app has got necessary privileges as per the security model and its configuration document declaration. If allowed to create the web view, the client app loads authorization server URL in the web viewers requesting the access token. With this step, the login page of the authorization server gets loaded within the web view. If user grants the access the authorization server redirects the web view to already registered redirect. W intercepts the redirection from the authorization server. W fetches the URL U authorization server has redirected to and compares the same against the list of the redirection URLs declared inside the configuration document of the underlying packaged web app. W retrieves from database the list S of redirection URLs that it created during the installation of the app. Next it compares U against each string entry u in S .

Given three finite-length strings a , b and c created using characters from an alphabet Σ of finite number of characters, i.e., $a, b, c, \Omega, \Sigma^*$. We denote equality of two such strings x and y as $x = y$. If all the entries in S fail to satisfy W aborts the OAuth execution process. If for any entry u in S , W finds $u = U$ satisfying, W stops checking remaining entries in S , if any, and extracts the last part β of the path

OAuth 2.0*Figure 13. Installation/Update flow for Parsing OAuth Element*

component of the redirection URL U . W matches β (last part of re) with app_id of the destination app using (6) to ensure if the OAuth is getting executed to the right, legitimate app. Upon successful verification of $\beta = app_id$, W extracts the access token ξ from U by fetching the value of the query parameter $access_token$ in U . W passes the access token ξ to packaged app for further use.

OAuth 2.0**Listing 3: OAuth flow control algorithm implemented by W.**

```

OAuthFLOWCONTROL() // OAuth Flow Controlled by WRT
1  w = a packaged web app; // Input
2   $\xi$  = OAuth access token to be used by w; // Output
3   $S = \emptyset$ ; // set of redirection URIs for each redirection URL u in W.configXml
4   $S = S \cup \{u\}$ ;
5  v = CREATE-OAuth-WEBVIEW(auth_server_url);
6  U = INTERCEPT-REDIRECT-RESP(); // redirect URL
7  if (true == MATCH-REDIRECT-URI(U, S))
8  D = PARSE-URI-FOR-APP-ID(U); // get app id
9  if (D == w.app_id)
10  $\xi$  = PARSE-URI-FOR-ACCESS-CODE(U);
11 CLOSE-OAuth-WEBVIEW(v);
12 return  $\xi$ ;
13 else
14 // OAuth flow terminated by WRT
15 OAuth-EXCEPTION(Target_Mismatch);
16 else
17 // OAuth flow terminated by WRT
18 OAuth-EXCEPTION(Redirection_Mismatch);

```

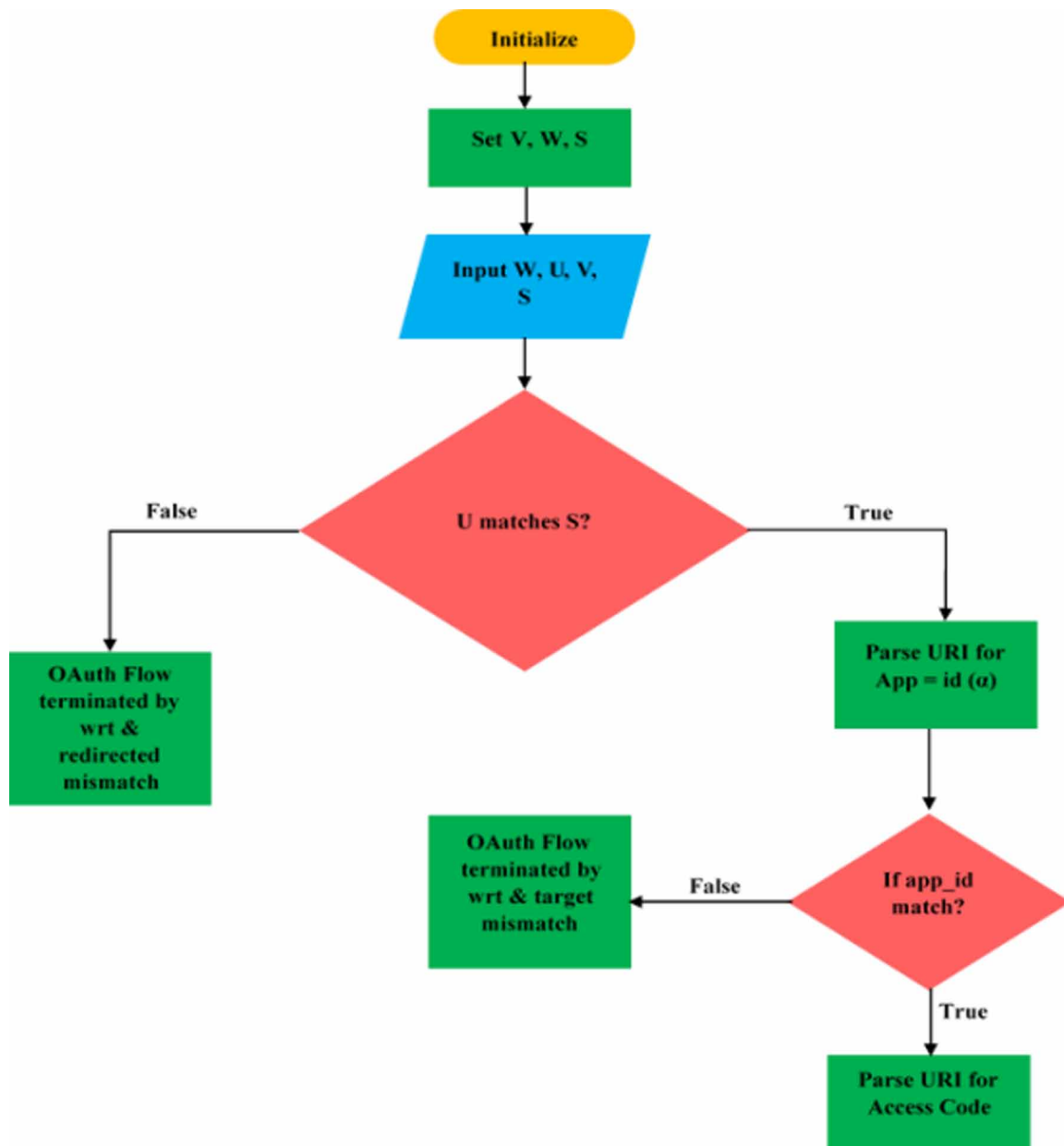
It does not satisfy for β with app_id, W concludes the current OAuth flow as illegal and aborts the OAuth execution flow raising exception to the packaged web app. In either case W closes the web view created for OAuth flow execution. Listing 3 shows the pseudocode for the algorithm followed by W to ensure secure OAuth flow initiated by a packaged web app. Figure 14 illustrates the flow chart of OAuth flow control using W.

Proposed Attack Model to Detect the Security of OAuth Flow

It occupied that 1) customer's browser and computer are impervious 2) both RP (Relying Parties) and IdP (Identity Provider) are not harmful, and 3) the forthright contacts between RP and IdP are assured. However, owing to the universality of web vulnerabilities (Hammer., 2010), IdP or RP may have several vulnerabilities, such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF). The objective of an invader is to reach an unauthorized access customer's funds hosted on IdP or RP. In concise, the power of the invader is restricted as follows:

- The charger is a web user who could consecrate a website, issue some links by the blog, microblog, and email, and start an attack on an attackable website.
- The invader is even a smaller who could smell the unencrypted traffic on the Internet, and bribe simply with the traffic.

Our analytical framework is to give a normal approach to find the security of OAuth service. It is a two-stage procedure adding protocol analysis and empirical analysis, as manifested in Figure 15. In this

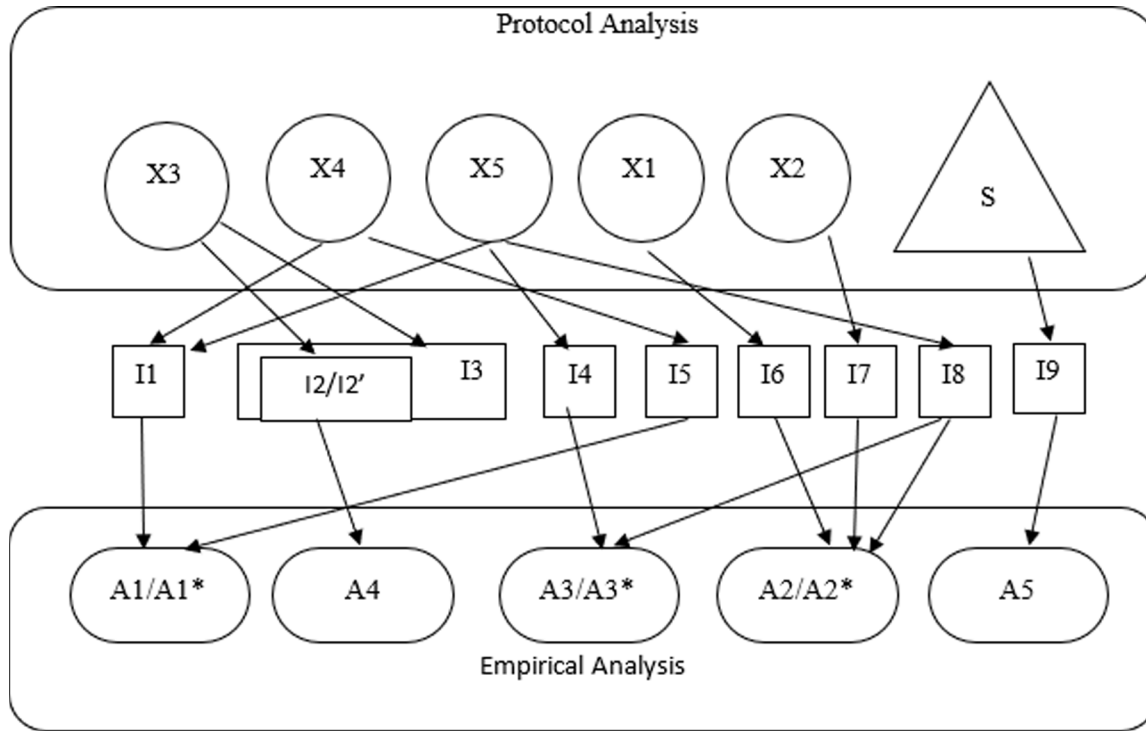
OAuth 2.0*Figure 14. Flow chart of OAuth flow control using W*

period of protocol analysis, we investigate the protocol gingerly to publish the usages, requirements and strong jeopardies of the five alterable parameters and session S.

Five attacks (recognized by A1 to A5) (lhshaoren., 2010) in view of the previous are given in the last stage, all of which have been demonstrated accessible in the accompanying test. What's more, the investigation on the presuppositions of the five assaults uncovers that we require just to execute a discovery on thing I1 to I9 as takes after to assess the security of OAuth services:

OAuth 2.0

Figure 15. Analytic framework of detecting the security on parameters and session S



- I1: Whether HTTPS protection is conveyed by RP.
- I2: Whether an unusual state parameter is utilized by RP.
- I3: Whether RP is resisted against CSRF attack.
- I4: Whether RP stores the access token in the URI or cookie.
- I5: Whether the code used in the cross.
- I6: Whether IdP underpins the two reactions composes all the while and unpredictably.
- I7: Whether any redirection URI in the domain of RP could pass IdP's checking.
- I8: Whether the token is a bearer token.
- I9: Whether a strategy to end the session S is provided.

Based on the analytic framework, the proposed approach can be described with the pseudocode as follows:

Listing 4: Pseudo-code for Attack List Detection on Parameters and Session S

```

Attack List Detection on Parameters and Session S
AttackList Detection() {
// Parameters X[1-5] and S are defined in Section 3.3;
// Attacks A[1-5] are defined in Section 3.4;
// The symbol * means the attack has greater damage. AttackList attacks = NULL;
// all the potential attacks. if(CHECK(X4.I1: HTTPS is employed by RP) = TRUE)

```


OAuth 2.0

```

{
if(CHECK(X4.I5: The code is cross in use) = TRUE) attacks.Add(A1*);
else
attacks.Add(A1);
}
if(CHECK(X1.I6: Two response types are supported by IP) = TRUE AND CHECK(X5.
I8: The token is a bearer token) = TRUE)
{
if(CHECK(X2.I7: The realm URIs are checked out by IP)= TRUE) attacks.Add(A2*);
else
attacks.Add(A2);
}
if(CHECK(X5.I4: The access token is stored incorrectly by RP) = TRUE
ANDCHECK(X5.I8: The token is a bearer token) = TRUE)
{
if (CHECK(X5.I1: HTTPS is employed by RP) = FALSE) attacks.Add(A3*);
else
attacks.Add(A3);
}
if(CHECK(X3.I3:Protection against CSRF is employed by RP) = FALSE)
{
attacks.Add(A4); // check Item X3.I2': Whether state is invalid. if(CHECK(X3.
I2: The state parameter is used by RP)= TRUE)
Warning(The state parameter(X3) is INVALID);
}
if(CHECK(S.I9:Ending session S is provided by RP)= FALSE) attacks.Add(A5);
return attacks;
}

```

PERFORMANCE ANALYSIS OF NEW OAuth SECURITY ARCHITECTURE FOR PACKAGED WEB APPS

When we make our Packaged Web application, we marked the following options, which all used the Web-native languages like HTML5, CSS, and JavaScript. We follow the Web application life-cycle from commodity concept via upliftment and redemption to end-of-life application retirement.

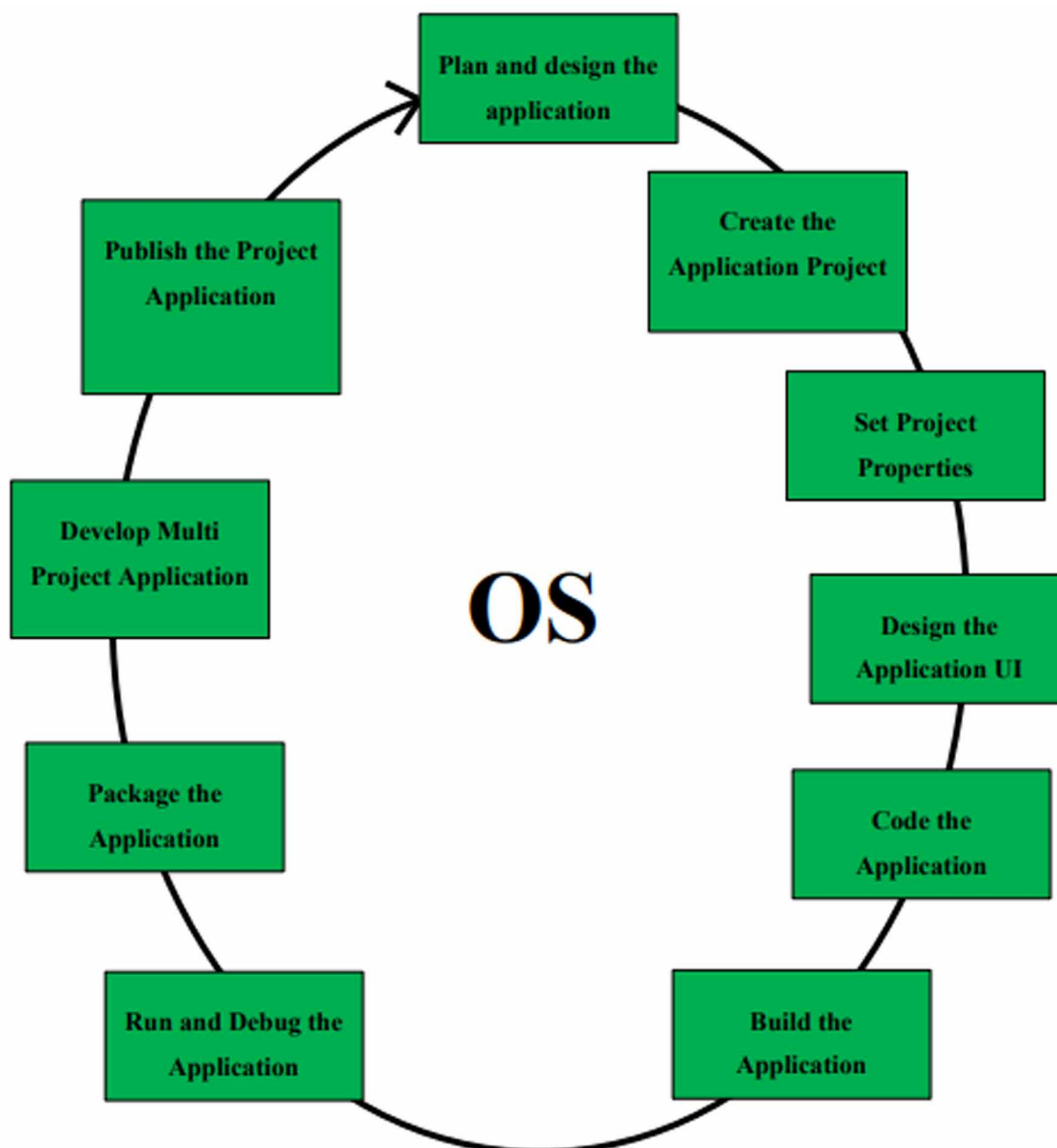
Planning and Designing the Application

The first step in creating a Web application is scheming and designing it using the design instruments of our option. Sometimes we have closed the application scheme and design, we are prepared to make the application project.

Creating the Application Project

We create our application using Visual Studio 2012. The Visual Studio gives several project templates that create it easier for us to initiate coding our application. When we make a new project, we can choose an earmarked template or specimen. Founded on the choice, the Visual Studio Web Project Wizard automatically builds basic functionalities that the application must apply to be capable to run. The default project files and folders are also made.

Figure 16. Packaged Web application development process



OAuth 2.0

Setting Project Properties

After making the application project, we may configure the characteristics of the project and the Web application to acquire the need functionality and properties for our application.

Designing the Application UI

We designed the application UI utilizing the UI materials defined in the Visual Studio Advanced UI Framework. It gives tools, like effects, UI components, events, and animations, for Web application development. We leverage these tools by just selecting the required screen elements and creating the applications.

Coding the Application

The Visual Studio Web Device API based on JavaScript gives advanced ingress to the device's platform capabilities. We can develop rich Web applications using the Visual Studio Web Device APIs. We can, for example, control the application life-cycle, manage our schedules, exchange data, or make payments using NFC. Sometimes we have ended coding our application, we are ready to construct our application.

Building the Application

When the Visual Studio creates an application then the following procedure is executed:

- Validation check for:
 - JavaScript, CSS
 - Privilege
- Compile for:
 - Coffee script
 - Less

If the project has some errors, they are displayed in the Problems and Project Explorer scenes after the construct. Anyone can create a Web application automatically or manually. We build our apps manually. In the Visual Studio menu, select Project > Build Project. We may construct our project at our benefit. If we need to usage the manual set up and confirm that the Project > Build Automatically choice is not picked.

Running and Debugging the Application

We use Google Chrome to run our application. The running of our application illustrated pictorially in below:

When the Visual Studio runs or debugs the application, the following procedure is executed:

- Construct automatically if no construct has been made yet.
- Package: The optimization procedure is only executed when we execute the packaging procedure.
- Execute the application to the imitator or destination device.

OAuth 2.0**Comparison Table of Existing Approaches With Proposed Approach**

The comparison of existing approaches with a proposed approach is given below in table 3:

Packaging the Application

Our Web application packaging procedure is founded on the W3C packaging and configuration.

When the Visual Studio packages the application, the following procedure is executed:

- Construct automatically if no construct has been made yet
- Optimize resources:
- Obfuscation (for JavaScript)
- Minification (for HTML, CSS, JavaScript, and PNG)
- Make the frame formation (for hybrid core applications).
- Create resources (for font, hybrid core, and UI framework applications).
- Maintain signing

We may bunch a Web application behaving the web-packaging mandate in the Command Line Interface (CLI) and where it is a working instrument in the Visual Studio: web-packaging project.wgt project/. By default, the Web application bundle is made at one time. We may display the pack volume at any drop of the application development procedure by double-clicking the project.wgt sfile in the Project Explorer display. All the files existent in the application project are viewed in a chart. *Certifying and Publishing the Application*

After we have packaged our application, we are ready to certify and publish our application. To certify and publish our application:

- Upload our Web application to the localhost.
- After the application is uploaded then the application is signed as an attested application installer bundle and the <Application_name>.wgt archive ordering, which holds the distributor signature and it is attached by the applicable hoard.

If we need to pick off your application from parceling and action, you want to request for application retirement from the hoard.

Comparison Between Existing Approaches With Proposed Approach

In our experimental setup, we compared our approach with two other approaches for OAuth flow execution – Approach A: the client app runs a web server, and Approach B: the client app creates a child window and initiates OAuth flow from the child window. In Approach A the OAuth access token is received by the web server and in Approach B the client keeps polling the HTTP redirect by running a timer with 100 ms timeout. 100 ms timeout has been considered after several trials optimizing the performance for OAuth response handling using Approach

B. In timer callback the client app checks if it has received a valid HTTP redirect with access code. If not, it again starts the timer with 100 ms timeout and continues waiting till the next timeout.

OAuth 2.0

This process is repeated until the client app receives a valid HTTP redirect with access code in it or the number of attempts exceeds a pre-determined number. If the client app receives a valid HTTP redirect with the access code in it, it closes the child window and makes an AJAX request to the authorization server to get the access token. The client app uses the access code received in the response to its earlier AJAX request. We used packaged web app for accessing resources from Facebook. We also created malicious versions of these apps impersonating the actual apps. We compared the system behavior while executing OAuth flow from legitimate and malicious apps. We also measured the time to handle OAuth redirection response and average power consumed during OAuth flow execution.

Characteristics

We have some common characteristics among the approaches. They also have some different characteristics. Now we mention the characteristics of the approaches:

- Supported Platform
- CSRF Attack
- XSS Attack
- Illegal OAuth Flow
- Response Handling Delay
- Power Consumption

Table 3. Comparison table of existing approaches with proposed approach

Parameter	Approach A	Approach B	Proposed Approach
Supported Platform	Windows 7,8,10, MAC OS, iOS 6.0 and later and Android 2.0 and later	Windows 7,8,10, MAC OS, iOS 6.0 and later and Android 2.0 and later	Windows 7,8,10, MAC OS, iOS 6.0 and later and Android 4.0 and later
CSRF Attack	Suffer Slightly	Suffer Mostly	Don't Suffer
XSS Attack	Suffer Slightly	Suffer Mostly	Don't Suffer
Illegal OAuth Flow	Some of these are blocked	Some of these are blocked	Aborts All
Response Handling Delay	Medium	High	Low
Power Consumption	Medium	Worst	Low

Performance Analysis and Results

In this section, we compared our approach with two other approaches for OAuth flow execution –Approach A: the client app runs a web server, and Approach B: the client app creates a child window and initiates OAuth flow from the child window. We compared the system behavior while executing OAuth flow from legitimate and malicious apps. We also measured the time to handle OAuth redirection response and average power consumed during OAuth flow execution. The OAuth flow is executed from our apps in 5 iterations.

OAuth 2.0

System Setup

In this section, the application will be examined. Further, the testing environment is described.

- Application Tools: Microsoft Visual Studio 2012, Microsoft SQL Server and Sublime Text is used to create the application.
- Testing Environment: The specifications of the computer used are:
- Operating System – Windows 7.
 - Processor – Intel(R) Core (TM) i7-6700 CPU @ 3.40GHz.
 - RAM – 16 GB (DDR-3).
 - Clock – Core speed 3401 MHz, 4 Core(s)
 - Bandwidth – 100 mbps

Analysis

In this section, the performance of new OAuth security architecture for Packaged Web apps is described with the corresponding figure, table and performance graph. At the initial stage, we analyze the security of our new OAuth security architecture for Packaged Web apps with the proposed attack model to detect the security of OAuth flow described in section 5.6. Five types of attacks (identified by A1 to A5) founded on the previous are given in the latter period, all of which have been verified available in the following research. And the resolution on the prolepsis of the five attacks publishes that we want only to execute a discovery on item I1 to I9.

Here we used DVWA (Damn Vulnerable Web App) web app tools to perform the previously defined attacks. Some of the attacks are listed below:

- Access token eavesdropping (A1)
- Access token theft via XSS (A2)
- Impersonation (A3)
- Session swapping (A4)
- Force-login CSRF (A5)

We have used the Website plus Tool for measuring the response handling delay (ms) for OAuth flow. We perform 5 iterations with our new developed Packaged Web app. Figure 18 to Figure 22 pictorially shows the iteration results for measuring the response handling delay.

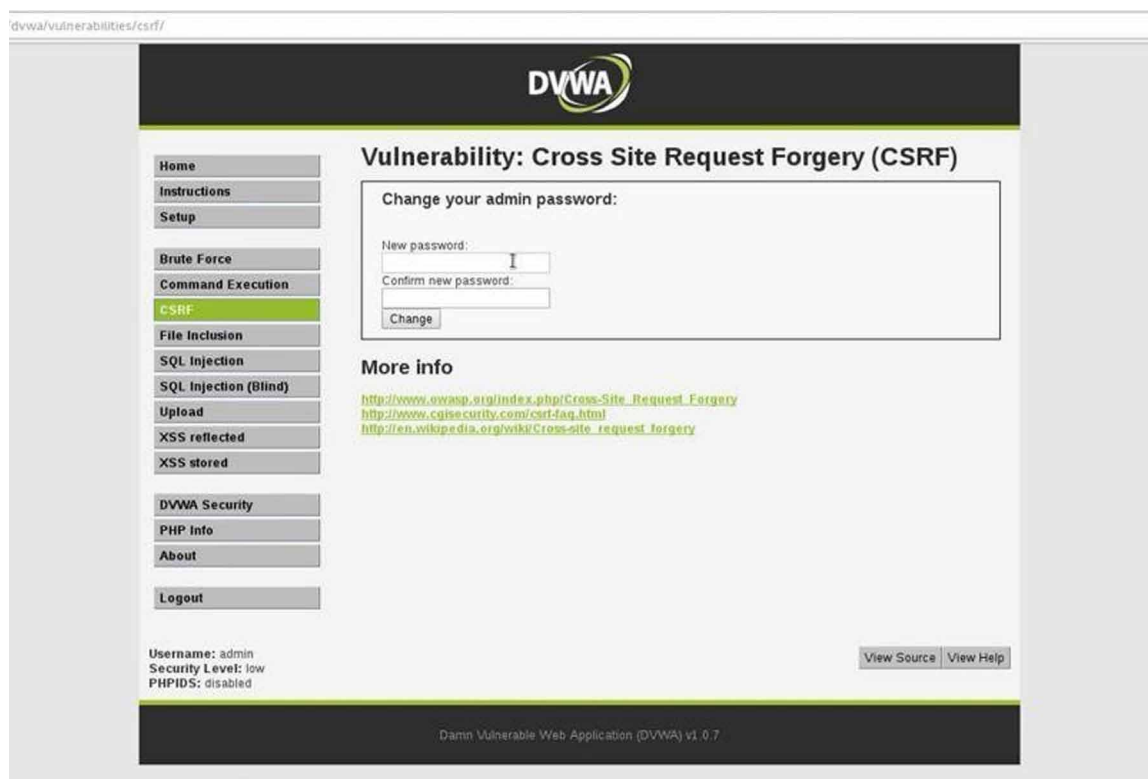
We can calculate the Response Handling Delay as:

$$T_{RHD} = T_{CBF} - T_{ARL}$$

Here we denote TRHD as a time for response handling delay, TCBF as time for call back first and TARL as time for OAuth request last. By using the above equation, we got 53.3 ms as response handling delay from the first iteration. Then we got 49.7 ms as response handling delay from the second iteration.

We got 50.8 ms as a response handling delay from the third iteration.

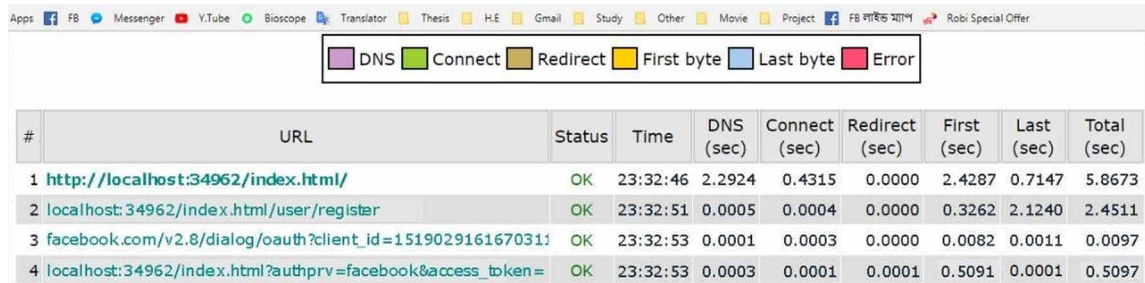
We got 53.9 ms as a response handling delay from the fourth iteration.

OAuth 2.0*Figure 17. Attacks tool - Damn Vulnerable Web App**Figure 18. Capturing response handling delay (Iteration 1)*

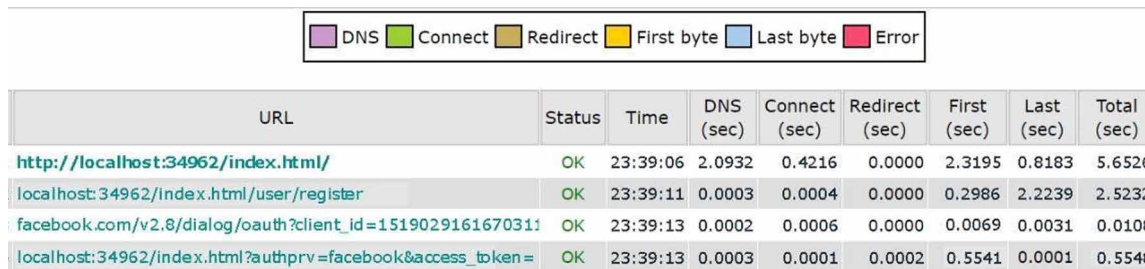
<div> DNS Connect Redirect First byte Last byte Error </div>									
#	URL	Status	Time	DNS (sec)	Connect (sec)	Redirect (sec)	First (sec)	Last (sec)	Total (sec)
1	http://localhost:34962/index.html/	OK	23:21:07	2.5258	0.2273	0.0000	2.3617	0.9176	6.0324
2	localhost:34962/index.html/user/register	OK	23:21:13	0.0000	0.0000	0.0000	0.2307	1.4495	1.6802
3	facebook.com/v2.8/dialog/oauth?client_id=1519029161670311	OK	23:21:15	0.0004	0.0006	0.0000	0.0095	0.0059	0.0167
4	localhost:34962/index.html?authprv=facebook&access_token=	OK	23:21:15	0.0003	0.0007	0.0000	0.5329	0.0001	0.5341

Figure 19. Capturing response handling delay (Iteration 2)

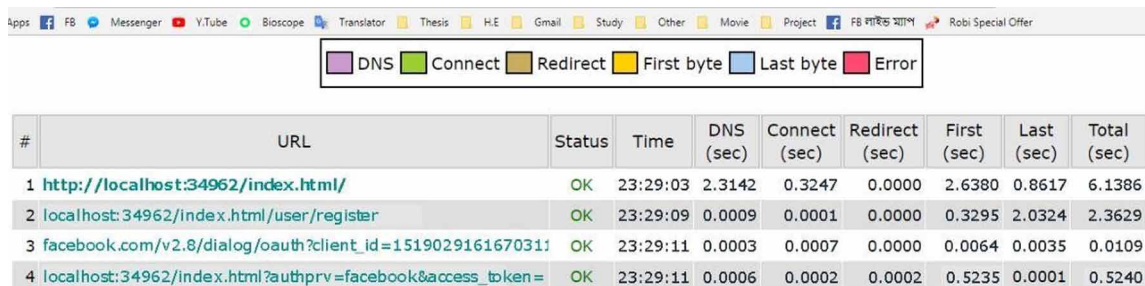
<div> DNS Connect Redirect First byte Last byte Error </div>									
#	URL	Status	Time	DNS (sec)	Connect (sec)	Redirect (sec)	First (sec)	Last (sec)	Total (sec)
1	http://localhost:34962/index.html/	OK	23:24:21	2.5008	0.2273	0.0000	2.3617	0.9176	6.0074
2	localhost:34962/index.html/user/register	OK	23:24:27	0.0001	0.0000	0.0000	0.2307	1.4495	1.6802
3	facebook.com/v2.8/dialog/oauth?client_id=1519029161670311	OK	23:24:29	0.0006	0.0004	0.0000	0.0095	0.0020	0.0125
4	localhost:34962/index.html?authprv=facebook&access_token=	OK	23:24:27	0.0002	0.0008	0.0000	0.4990	0.0001	0.5001

OAuth 2.0*Figure 20. Capturing response handling delay (Iteration 3)*


#	URL	Status	Time	DNS (sec)	Connect (sec)	Redirect (sec)	First (sec)	Last (sec)	Total (sec)
1	http://localhost:34962/index.html/	OK	23:32:46	2.2924	0.4315	0.0000	2.4287	0.7147	5.8673
2	localhost:34962/index.html/user/register	OK	23:32:51	0.0005	0.0004	0.0000	0.3262	2.1240	2.4511
3	facebook.com/v2.8/dialog/oauth?client_id=1519029161670311	OK	23:32:53	0.0001	0.0003	0.0000	0.0082	0.0011	0.0097
4	localhost:34962/index.html?authprv=facebook&access_token=	OK	23:32:53	0.0003	0.0001	0.0001	0.5091	0.0001	0.5097

Figure 21. Capturing response handling delay (Iteration 4)


URL	Status	Time	DNS (sec)	Connect (sec)	Redirect (sec)	First (sec)	Last (sec)	Total (sec)
http://localhost:34962/index.html/	OK	23:39:06	2.0932	0.4216	0.0000	2.3195	0.8183	5.6526
localhost:34962/index.html/user/register	OK	23:39:11	0.0003	0.0004	0.0000	0.2986	2.2239	2.5232
facebook.com/v2.8/dialog/oauth?client_id=1519029161670311	OK	23:39:13	0.0002	0.0006	0.0000	0.0069	0.0031	0.0108
localhost:34962/index.html?authprv=facebook&access_token=	OK	23:39:13	0.0003	0.0001	0.0002	0.5541	0.0001	0.5548

Figure 22. Capturing response handling delay (Iteration 5)


#	URL	Status	Time	DNS (sec)	Connect (sec)	Redirect (sec)	First (sec)	Last (sec)	Total (sec)
1	http://localhost:34962/index.html/	OK	23:29:03	2.3142	0.3247	0.0000	2.6380	0.8617	6.1386
2	localhost:34962/index.html/user/register	OK	23:29:09	0.0009	0.0001	0.0000	0.3295	2.0324	2.3629
3	facebook.com/v2.8/dialog/oauth?client_id=1519029161670311	OK	23:29:11	0.0003	0.0007	0.0000	0.0064	0.0035	0.0109
4	localhost:34962/index.html?authprv=facebook&access_token=	OK	23:29:11	0.0006	0.0002	0.0002	0.5235	0.0001	0.5240

We got 52.0 ms as a response handling delay from the fifth iteration. The average response handling delay of our new developed Packaged Web app is 51.94 ms.

Then we used the Joulemeter Tool for measuring the power consumption (mW) for OAuth flow. We perform 5 iterations with our new developed Packaged Web app. Figure 23 pictorially shows the iteration results for measuring power consumption.

We can calculate the Response Handling Delay as:

$$E_{(mW/time)} = P_{(nW)} * t_{(time)} / 1000_{(W/mW)}$$

Here we denote E mW/time as energy, P mW as power and t as time.

We got 173 mW as power consumption from the second iteration. Then we got 175 mW as power consumption from the third iteration. We got 170 mW as power consumption from the fourth iteration.

OAuth 2.0*Figure 23. Capturing power consumption (Iteration 1 to 5)*

And then we got 175 mW as power consumption from the fifth and final iteration. The average power consumption of our new developed Packaged Web app is 173 mW.

Handling Illegal OAuth Flow

In this section we discuss handling capacity of our new OAuth based Packaged Web Application. Since our prime goal was to check if our approach can block malicious OAuth, we first checked the behavior with malicious OAuth flow initiated by impersonating client web apps. All these malicious web apps create OAuth web view and attempt OAuth flow using app id of some other legitimate packaged web app. Results reveal that our approach blocks all malicious OAuth flow executions which are summarized in table 4. Since in our approach W checks the app id present in the redirection URL against the app id the app executing the OAuth flow W aborts OAuth flows from malicious impersonating apps considering those flows as illegal flows. Since an app cannot tamper its app id and W also not allow it to use app id of an already installed legitimate app, W always successfully aborts all illegal OAuth flows.z

Table 4. OAuth Flow execution results from our apps

Attack No.	Attack Type	Resource Server	Attacks (Attempted/ Blocked)
A1	Access token eavesdropping	Facebook	1/1
A2	Access token theft via XSS		3/3
A3	Impersonation		1/1
A4	Session swapping		2/2
A5	Force-login CSRF		5/5

The algorithm OAuthFLOWCONTROL() executed by W is the key component of our approach. In the other two approaches, there is no means to check if the access token is getting delivered to right, intended, a legitimate target. But unlike, those two approaches by means of this algorithm our method ensures that access token if delivered, is delivered to the right, intended legitimate target. W ensures that by executing this algorithm OAuthFLOWCONTROL().

OAuth 2.0**Evaluation Table of OAuth Response Handling Delay of Proposed Approach**

Our approach gives better OAuth response handling delay compared to the other two approaches across all five iterations. In the case of Approach A, first the authorization code is received and then the access token is received next. Hence, getting access token requires a greater number of message exchanges with the authorization server and hence its OAuth response handling delay is more compared to our approach. The Evaluation Table of OAuth Response Handling Delay of Proposed Approach is given in Table 5:

Table 5. Evaluation table of OAuth response handling delay of proposed approach

Iteration No.	Resource Server	Response Handling Delay (ms)	Average Response Handling Delay (ms)
1	Facebook	53.9	52.94
2		53.3	
3		49.7	
4		52.0	
5		50.8	

Graphical Representation of Response Handling Delay of Proposed Approach

The graphical representation of Response Handling Delay of the proposed approach is given below

Graphical Representation of Response Handling Delay with Existing Two Approaches

The graphical representation of response handling delays with existing two approaches is given below:

Evaluation Table of Power Consumption of Proposed Approach

Since three approaches handle OAuth flow involving different software and hardware components, we thought of capturing the average power consumption during OAuth flow execution with each of these approaches. These approaches possess the access token by executing different techniques and require additional processing to gain the possession of the access token, employing OAuth during execution web app will cause additional CPU overhead. Therefore, power consumption by web apps due to OAuth execution is a key performance indicator to compare OAuth handling techniques. The Evaluation Table of OAuth Power Consumption of Proposed Approach is given in table 6:

Graphical Representation of Power Consumption of Proposed Approach

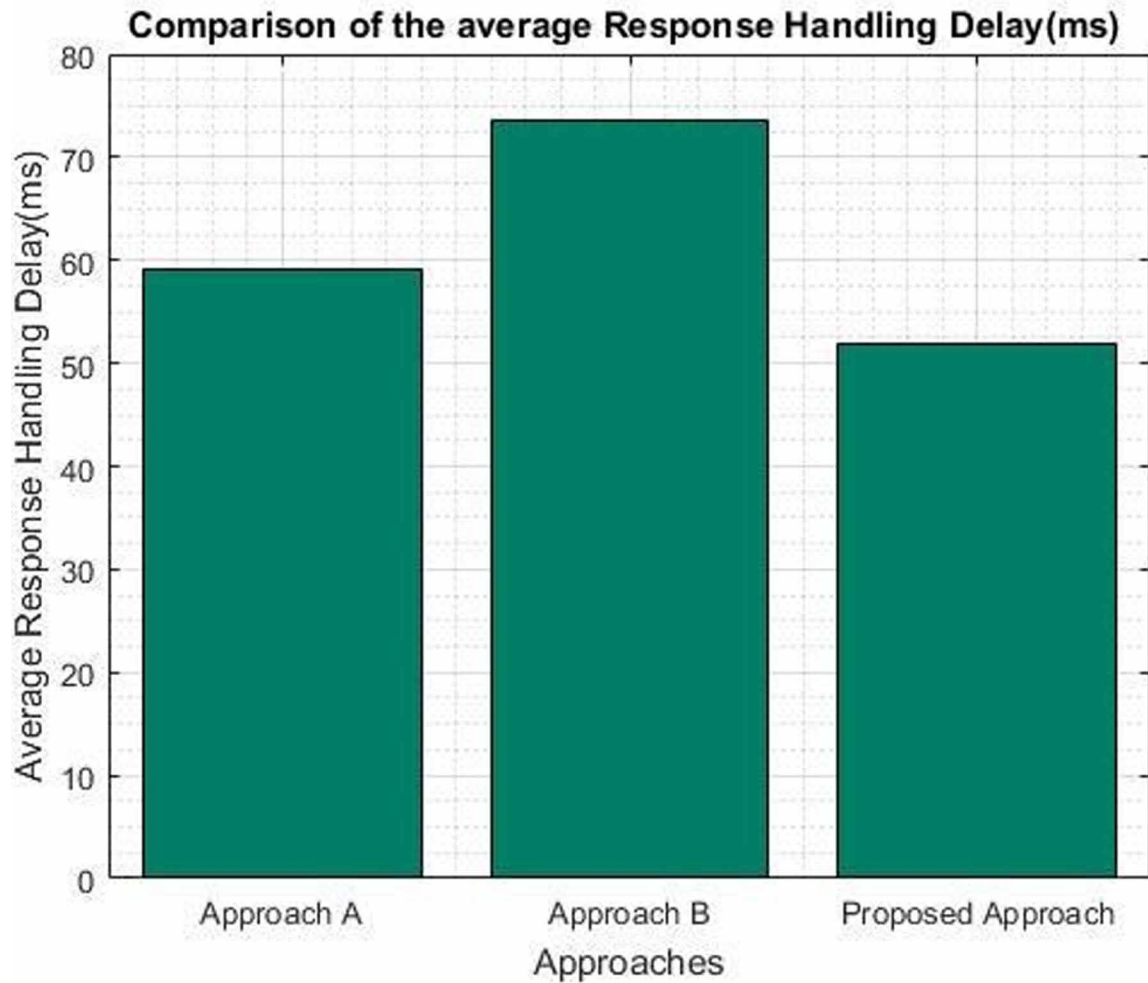
The graphical representation of power consumption of the proposed approach is given below for all five iterations:

OAuth 2.0*Figure 24. Graphical representation of response handling delay of proposed approach***Graphical Representation of Power Consumption with Existing Two Approaches**

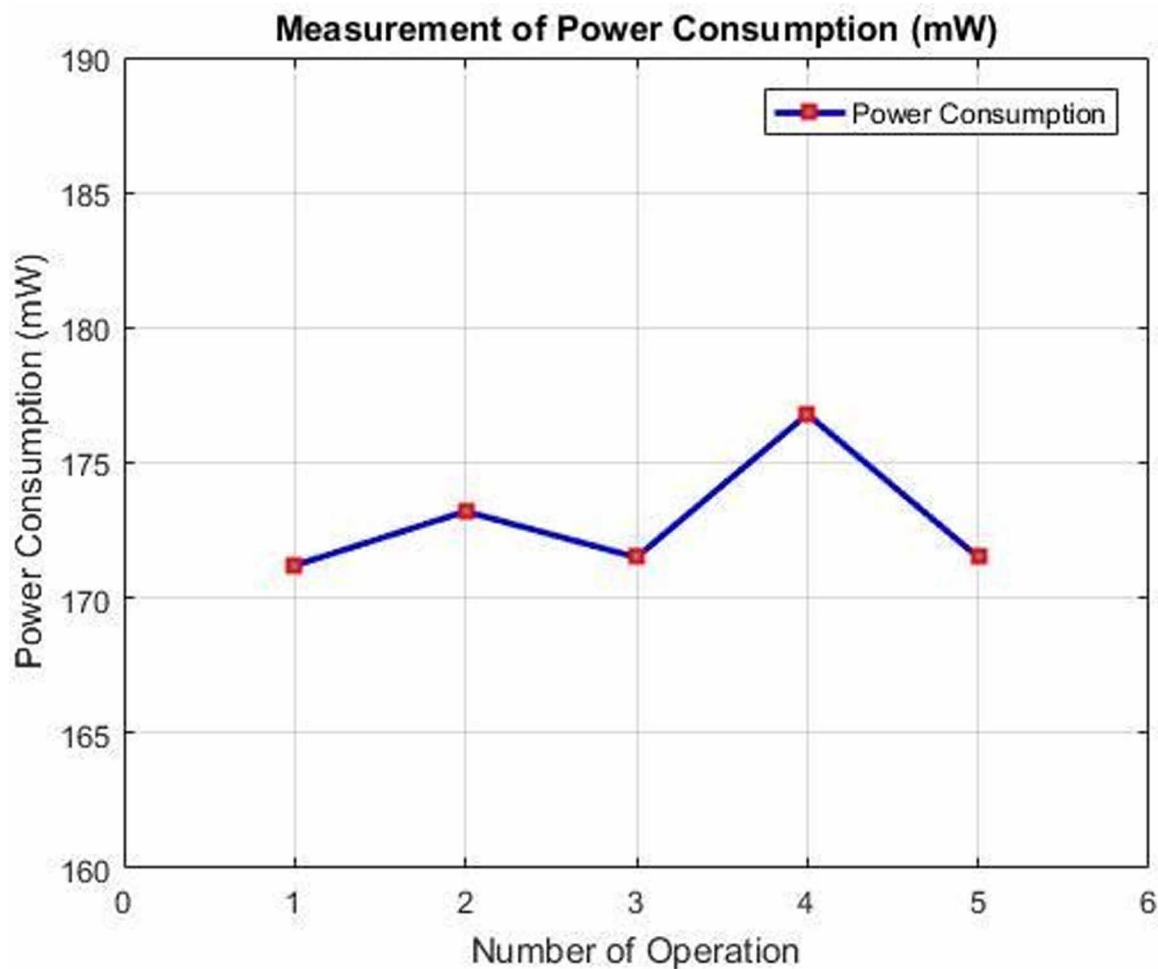
The graphical representation of power consumption with existing two approaches is given below:

CONCLUSION AND FUTURE WORK

Unlike operable security protocols, OAuth 2.0 is designed except sound cryptographic protection like a digital signature, a random nonce, and encryption. The deficiency of encryption in the protocol needs RPs to employ SSL, but many appreciated websites do not follow this exercise. Compared to server stream (Adrienne, P. F., Elizabeth, H., Serge. E., Ariel, H., Erika, C. & David, W., 2012) client-flow is inherently insecure for OAuth. Based on these acumens, we trust that OAuth 2.0 at the hand of maximum developers without a profound understanding of web security is perhaps to produce precarious implementations.

OAuth 2.0*Figure 25. Graphical representation of response handling delay with existing two approaches**Table 6. Evaluation table of OAuth power consumption of proposed approach*

Iteration No.	Resource Server	Power Consumption (mW)	Average Power Consumption (mW)
1	Facebook	171	173
2		173	
3		175	
4		170	
5		173	

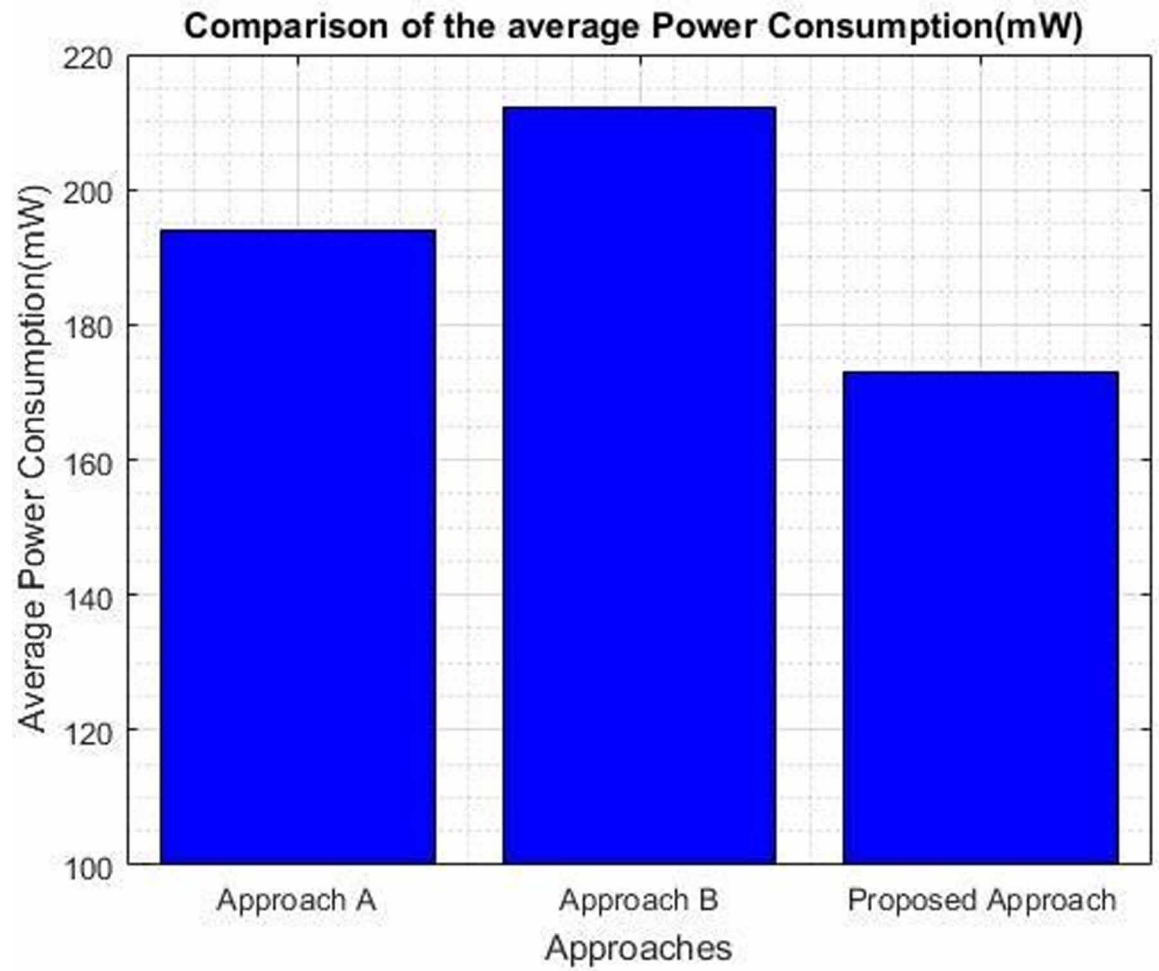
OAuth 2.0*Figure 26. Graphical representation of power consumption of proposed approach*

In our system any malicious app cannot execute OAuth flow impersonating as a legitimate app even after gaining the access to proper client_id, redirect_uri and other secrets this is ensured because the OAuth verification logic is executed by W that looks for the presence of the app id in the redirect URL. App id of a packaged app is unique across all apps and any app cannot tamper that. Hence, the strategy of embedding the app id in the redirect URL ensures that the redirect response will reach only to the legitimate app even though the app cannot define the redirect endpoint using a valid HTTP URL like a website.

Our approach still requires the developers to generate the redirection URLs to be registered and declared in the configuration document manually. But that opens the scope of manual errors, which if present, will stop a legitimate valid app from getting OAuth access token. In the future, we intend to wire logic of providing client app credentials through app stores reducing the possibility of developer errors. We also intended to use the Encryption method in future.

OAuth 2.0

Figure 27. Graphical representation of power consumption with existing two approaches



OAuth 2.0**REFERENCES**

Adam, B., Juan, C., & Dawn, S. (2009). Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. *IEEE Symposium on Security and Privacy*, 30(8), 360-371. doi: 10.1109/SP.2009.3

Adrienne, P. F., Elizabeth, H., Serge, E., Ariel, H., Erika, C., & David, W. (2012). Android permissions: user attention, comprehension, and behavior. *Eighth Symposium on Usable Privacy and Security*, (3). doi: 10.1145/2335356.2335360

Android WebView. (2018). Retrieved from <https://developer.android.com/reference/android/webkit/WebView.html>

Caimei, W., Yan, X., Wenchao, H., Huihua, X., Jianmeng, H., & Cheng, S. (2017). A Verified Secure Protocol Model of OAuth Dynamic Client Registration. *3rd International Conference on Big Data Computing and Communications (BIGCOM)*, 106 – 110. doi: 10.1109/BIGCOM.2017.50

Charlie, C., Ben, L., Benjamin, G. Z., & Christian, S. (2011). ZOZZLE: Fast and precise in- browser JavaScript malware detection. *Proceedings of the 20th USENIX Conference on Security*. Retrieved from <https://usenix.org/conference/usenix-security-11/zozzle-fast-and-precise-browser-javascript-malware-detection>

Daniel, F., Ralf, K., & Guido, S. (2016). A Comprehensive Formal Security Analysis of OAuth 2.0. *CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 1204-1215, doi: 10.1145/2976749.2978385

Dongwan, S., Huiping, Y., & Une, R. (2013). Supporting visual security cues for WebView-based Android apps. *28th Annual ACM Symposium on Applied Computing, ser. SAC '13. New York*, 1867–1876. doi: 10.1145/2480362.2480709

Dongwan, S., & Rodrigo, L. (2011). An empirical study of visual security cues to prevent the ssl stripping attack. *27th Annual Computer Security Applications Conference, ser. ACSAC '11. New York*, 287–296. doi: 10.1145/2076732.2076773

Elements, C. - W3C Working Group Note. (2018). Retrieved from <https://www.w3.org/TR/custom-elements>

Erika, C., & David, W. (2014). Bifocals: Analyzing webview vulnerabilities in android applications. *Information Security Applications - 14th International Workshop, WISA*, 8267. doi: 10.1007/978-3-319-05149-9_9

Facebook App in Tizen. (2018). Retrieved from <https://developer.tizen.org/ko/development/articles/facebook-apptizen>

Fadi, M., & Mohamed, S. (2016). Hardening the OAuth-WebView Implementations in Android Applications by Re-Factoring the Chromium Library. *2nd IEEE International Conference on Collaboration and Internet Computing*, 196–205. doi: 10.1109/CIC.2016.036

Francisco, C., & Karen, P. L. (2011). *Security Analysis of Double Redirection Protocols*. Retrieved from <https://bit.ly/2o5eggP>

OAuth 2.0

Google Identity Platform. (2018). *Using OAuth 2.0 to Access Google APIs*. Retrieved from <https://developers.google.com/identity/protocols/OAuth2>

Hammer. (2010). The OAuth 1.0 protocol. *The Internet Eng. Task Force RFC 5849*. Retrieved from <https://tools.ietf.org/html/rfc5849>

Hao, H., Vicky, S., & Wenliang, D. (2013). On the effectiveness of API-level access control using bytecode rewriting in Android. *8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 25–36. 10.1145/2484313.2484317

Hardt. (2011). The OAuth 2.0 authorization framework. *The Internet Eng. Task Force RFC 6749*. Retrieved from <https://tools.ietf.org/html/rfc6749>

Hardt. (Ed.) (2012). *The OAuth 2.0 authorization framework*. IETF Std., RFC6749. doi:10.17487/RFC6749

Jason, B., Elie, B., Divij, G., & John, C. M. (2010). State of the Art: Automated Black-Box Web Application Vulnerability Testing. *IEEE Symposium on Security and Privacy*, 31(1), 332 - 345. doi:10.1109/SP.2010.27

Jones & Hardt. (2012). *OAuth 2.0: Bearer token usage*. Available: <https://tools.ietf.org/html/draft-ietf-OAuth-v2-bearer-23>

Kaushik, D., Prabhavathi, P., & Joy, B. (2017). Security Mechanism for Packaged Web Applications. *2017 IEEE International Conference on Web Services (ICWS)*. doi: 10.1109/ICWS.2017.72

Kitura. (2018). *OAuth 2.0 authentication with Facebook/Google*. Retrieved from <https://kitura.io/docs/authentication/fb-google-OAuth2.html>

lhshaoren. (2010). *A vulnerability on weibo.com (sina.com)*. Retrieved from <http://www.wooyun.org/bugs/wooyun-2010-039727>

Lodderstedt, McGloin, & Hunt. (2013). *OAuth 2.0 Threat Model and Security Considerations*. IETF Std., RFC6819. doi:10.17487/RFC6819

Marian, H., Markus, H., Susanne, W., & Matthew, S. (2014). Using personal examples to improve risk communication for security & privacy decisions. *32nd Annual ACM Conference on Human Factors in Computing Systems*, 2647–2656. doi: 10.1145/2556288.2556978

McGloin & Hunt. (2011). *OAuth 2.0 Threat Model and Security Considerations*. Retrieved from <http://tools.ietf.org/id/draft-ietf-OAuth-v2-threatmodel-00.txt/>

Mike, T. L., & Venkat, V. (2009). Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. *IEEE Symposium on Security and Privacy*, 30(1), 331-346. doi: 10.1109/SP.2009.33

Mohamed, S., & Fadi, M. (2014). Towards Enhancing the Security of OAuth Implementations in Smart Phones. *IEEE Third International Conference on Mobile Services*, 9–46. doi: 10.1109/MobServ.2014.15

Nazmul, H., Md, A. H., Md, Z. H., Md, H. I. S., & Shawon, R. (2018). OAuth-SSO: A Framework to Secure the OAuth-Based SSO Service for Packaged Web Applications. *IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, 12(1), 1575-1578. doi:10.1109/TrustCom/BigDataSE.2018.00227

OAuth 2.0

OpenID Authentication 2.0 - Final. (2007). Retrieved from https://openid.net/specs/openid-authentication-2_0.html

Prateek, S., David, M., & Livshits, B. (2011). ScriptGard: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. *ACM Conference on Computer and Communications Security*, 18(2), 601-614. doi: 10.1145/2046707.2046776

Richer, Mills, & Tschofenig. (2012). *OAuth 2.0 message authentication code (MAC) tokens*. Retrieved from <https://tools.ietf.org/html/draft-ietf-OAuth-v2-http-mac-00>

Rui, W., Shuo, C., & Xiaofeng, W. (2012). Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. *2012 IEEE Symposium on Security and Privacy*, 365-379. doi: 10.1109/SP.2012.30

San-Tsai, S., Kirstie, H., & Konstantin, B. (2012). Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures. *Computers & Security*, 31(4), 465–483. doi:10.1016/j.cose.2012.02.005

San-Tsai, S., & Konstantin, B. (2012). The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. *CCS '12 Proceedings of the 2012 ACM conference on Computer and communications security*, 378-390. doi: 10.1145/2382196.2382238

Suresh, C., Charanjit, J., & Arnab, R. (2011). Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*. Retrieved from <https://eprint.iacr.org/2011/526>

Tara, W., & Kori, I. Q. (2005). Gathering evidence: use of visual security cues in web browsers. *Proceedings of the Graphics Interface 2005 Conference*, 137–144. doi: 10.1145/1089508.1089532

Victoria, B. (2016). Characterization of web single sign-on protocols. *IEEE Communications Magazine*, 54(7). doi:10.1109/MCOM.2016.7514160

Wanpeng, L., & Chris, J. M. (2014). Security Issues in OAuth 2.0 SSO Implementations. *International Conference on Information Security*, 8783(1), 529-541. doi: 10.1007/978-3-319-13257-0_34

Web Runtime – Tizen developer. (2018). Retrieved from <https://developer.tizen.org/development/training/web-application/understanding-tizen-programming/web-runtime>

William, K., & Robertson, G. V. (2009). Static Enforcement of Web Application Integrity Through Strong Typing. *USENIX Security Symposium*, 18(6), 283-298.

Yacin, N., Prateek, S., & Dawn, S. (2009). Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 16(1), 1-20.