Anoop Vijayan

Extending OAuth2.0 for Kerberos-like authentication to avoid Internet phishing attacks

Master thesis in Mobile Technology and Business

Faculty of Information Technology

University of Jyvaskyla
Department of Computer Science and Information Systems

Author: Anoop Vijayan

Contact Information: anvijaya@jyu.fi

Supervisor(s): Timo Hamäläinen
Department of Computer Science and Information Systems
University of Jyvaskyla

Reviewer(s): Timo Hamäläinen
Department of Computer Science and Information Systems
University of Jyvaskyla

Title: Extending OAuth2.0 for Kerberos-like authentication to avoid Internet phishing attacks

Project: Master thesis in Mobile Technology and Business

Page count: 72

ABSTRACT

The combined use of OpenID and OAuth for authentication and authorization is gaining popularity day by day in Internet. Because of its simplicity to understand, use and robustness, they are used in many domains in web, especially where the apps and user base are huge like social networking. Also it reduces the burden of typing the password every time for authentication and authorization especially in hand-held gadgets.

After a simple problem scenario discussion, it is clear that the OpenID+OAuth combination has some drawbacks from the authentication perspective. The two major problems discussed here include problems caused due to transfer of user credentials over Internet and complexity in setting up of two protocols separately for authentication and authorization.

Both the problems are addressed by extending OAuth2.0. By using Kerberos-like authentication, the user has the possibility of not passing the credentials over Internet. It is worth to note that, OAuth2.0 also uses some kind of tokens for authorizations similar to Kerberos. It could be seen that extending OAuth2.0 to perform authentication removes the need for OpenID and its problems completely.

Keywords: OpenID, OAuth, Kerberos, Internet phishing, authentication

GLOSSARY

OpenID	Open standard authentication
OAuth	Open Authorization
OP	Open Identity Provider/OpenID provider/Identity Provider
RP	Relaying Party
SP	OAuth Service Provider
TCP/IP	Transmission Control Protocol/Internet Protocol
HTTP	Hypertext transfer protocol
URL	Uniform resource locator
AS	Authentication Server
TGS	Ticket Granting Server
TGT	Ticket Granting Ticket
KDC	Key Distribution Center
REQ	Kerberos based Request
RES	Kerberos based Response
UCS	Unicode character set
MIT	Massachusetts Institute of Technology
K-OAuth	Kerberos OAuth
JSON	JavaScript Object Notation
AJAX	Asynchronous JavaScript and XML
UTF-8	Unicode Transformation Format – 8-bit
AE	Authentication and Authorization entity
TLS	Transport Later Security
CC	Combined Consumer (RP+OAuth consumer)
CP	Combined Provider (OP+SP)
MAC	Media Access Control
XOR	Exclusive OR

ACKNOWLEDGEMENTS

During the course of this work, I have had assistance and support from many people. First and foremost, I wish to express my profound gratitude to Professor Timo Hamalainen for his highly proactive supervision of the research, decisive guidance, great attention and care, insightful comments, and extensive assistance. His reassuring approach permitted me to achieve the objectives which I set out for.

I want to express special gratitude to Jari Kellokoski for his guidance and support during the write up. His encouragement and support enabled me to come up with a paper that expresses my ideas fluidly, appropriately and in style, while conforming to the conventions of the Faculty of Information Technology at the University of Jyvaskyla.

I would like to thank the Faculty of Information Technology, especially the Mobile Technology and Business master programme coordinators and creators for giving me the opportunity to study at University of Jyvaskyla.

Finally, an honourable mention goes to our families, especially my wife and friends for their understandings and supports on us in completing this project. Without whom I would have faced many difficulties.

I am grateful to all the people of Faculty of Information Technology who either directly or indirectly enabled me to carry out this research.

Anoop Vijayan

Jyväskylä 2012

CONTENTS

List of Figures

1. Introduction

The act of confirming the truth of an attribute is defined as authentication. It could be considered as confirming the identity of a person. On the other hand authorization is the act of specifying access rights to resources or managing access control.

OpenID is an open standard performing authentication in a decentralized manner by consolidating user's digital identities [1]. Primarily avoiding the misuse of identity-related information and preventing and detecting identity theft in cyberspace, OpenID is a user-centric identity-usage that runs on a trusted third party [2]. It is a single sign-on (SSO) protocol, which can solve the above problems by using a single pair of user-id and password for different websites that support OpenID. Users can log onto the website with unique user-id (their email address or a URL) and this user-id is open to all the web applications in the Internet. The password can be centrally managed by OpenID Provider (OP). Thus, OP is responsible for users' information security. If OP is attacked, it will be a disaster for users [29]. In simple words a user could get authenticated through an Identity Provider (OP) to a website called Relying Party (RP) without even having a user account locally in the website. For instance, logging into a news website with a Facebook account and password.

OAuth is an open standard for authorization [3]. OAuth provides a method for clients to access server resources on behalf of a resource owner. It provides a process for end-users to authorize third-party access to their server resources without sharing their credentials using user-agent redirections [4]. This could be considered as an ability to comment an article in the news website example given above.

The combined usage of OpenID with OAuth has been gaining popularity because of its simplified usage especially with hand held devices. This combined framework brings benefits to all the roles involved in the system in a non-intrusive and user-centric way. Also such a system based on open technologies makes the composition of services easier and accelerates the on-boarding of service providers [5]. Mobile and handheld devices are evolving into hubs of content and context information. Therefore, focuses on pervasive applications in smart spaces that use

locally available connectivity and device discovery allow, sharing content and offering services locally with direct connections between devices [6]. This requires such a framework that integrates authentication and authorization seamlessly with the user experience.

1.1 Research Problem

The authentication mechanism used in OpenID, an HTTP-based URL authentication protocol would require passing credentials over TCP/IP [7]. This has high potential of Internet attacks like Internet phishing [8]. When the RP requests the OP to authenticate a user via a user browser, the malicious RP redirects the user to a phishing page with the same content provided by the OP. Then, the user enters the password assuming the page is provided by the OP. The malicious RP obtains the user's OpenID and password. Although a user can authenticate by password, the user cannot authenticate an OP. Thus, OpenID is vulnerable to attacks like phishing [9].

On the other hand, two different protocols are used for authentication and authorization making the setup complicated [10]. There are couple of problems related to RP adoptions which are worth mentioning. The first is the “NASCAR problem” where users must pick an OpenID from the many available options. The second issue is that the RP loses some control over its relationship with any given user or the associated identifying data that do not provide much incentive to service providers [11].

These problems could be solved by improving the authentication part of the process. This article discusses how OAuth, which is primarily used for authorization purposes, could be extended also to perform authentication. This solves the problem related to complexity mentioned above. After OAuth is capable of performing both authentication and authorization, there is a relatively simple and unified system. Attacks over Internet could be considerably reduced if passing of credentials is somehow circumvented [12].

Kerberos is a computer network authentication protocol which works on the exchanging of *tickets* to allow nodes communicating over a non-secure network to prove their identity to one another in a secure manner [13]. As OAuth also works with ticket hand outs, it could be extended to perform authentication similar to Kerberos [14].

By implementing the Key Distribution Center (KDC) to OAuth server, the authentication negotiations performed in Kerberos could be performed in the OAuth server itself. This would make a setup to perform both authentication and authorization in a single suite, thus overcoming those drawbacks mentioned above.

1.2 Related work

Considerable amount of research is performed to address the problems related OpenID. One such is the assurance ID, which refers to the identity check of users who request an ID provider to generate an account before it issues an Open ID to a user. Ordinary Internet services require only an e-mail address for generating user accounts. A user who holds a free mail address can generate user accounts on a service. It is difficult for the provider to find the real identity of the user. This could be overcome by Assurance ID by validating an officially recognized ID in the local region offline [30]. One of the biggest problems with OpenID is its vulnerability to Internet phishing attacks, a process of redirecting OpenID from a RP to an OP when users log in with the OpenID to use the OpenID service [15]. Not surprisingly, many studies were also performed to overcome such problems. Some of the studies involve addition of meta-authentication like using I-PIN to prevent RP phishing [15]. Also some involve usage of two types of passwords for anti-phishing. The password is divided into fixed password and temporary password. Fixed password used on PCs which will be bound and is appropriate for PCs that are used frequently by the user. Temporary password can be used on any PC, but its life cycle is short. Through analysis, this method can effectively avoid phishing [16]. Though there are some attempts to fight Internet phishing with tokens and authentication e-mail, the methods are based on the assumption that the number of OPs is small, and are hence safe from attackers and easier to realize from the technical viewpoint than existing methods [10]. This lays considerable limitation on scalability and increases complexity due to the two-factor authentication. The use of OpenID with OAuth combined suite for identity management has been also getting popular [17], [6]. OpenID provides the single sign-on feature. The user, who has been authenticated by the authentication server, can establish sessions with other servers. OAuth allows users to grant their access authorities to servers, which use the granted authorities when establishing new sessions with other servers. Several large Web sites have already introduced these technologies because they

are essential for permitting modern Web sites to interwork with each other by establishing sessions [24]. Usage of Kerberos protocol for authentication other than desktop is unusual; however, there have been some attempts [18], [19]. It's one of the distributed authentication system that allows a client to prove its identity to a server without sending data across the network that might allow an attacker to subsequently impersonate that principal. Kerberos can solve many of the security problems of large, heterogeneous networks, including mutual authentication between clients and servers. Extensions to Kerberos can provide for the use of public key cryptography during certain phases of authentication [31]. Lately, there have been also many attempts to expand OAuth for authentication also [20].

2. OpenID and Authentication

OpenID 1.0 was originally developed in 2005 by Brad Fitzpatrick, Chief Architect of Six Apart, Ltd. It is now deployed by a wide range of websites, particularly those heavy in user-generated content. As this user base evolved, the need for new OpenID Authentication features also increased. A community of individuals and companies including VeriSign, Inc., JanRain, Inc., Cordance Corporation, NetMesh, Inc., Six Apart, Ltd., and Sxip Identity, Inc. shared the vision that OpenID could become an umbrella under which multiple technologies can fit. They began collaborating to define the next version of OpenID Authentication and other specifications that built the OpenID framework. OpenID framework then evolved with Authentication 2.0 specification, a data transfer protocol to support both pull and push use cases and extensions to support the exchange of rich profile data and user-to-user messaging. The goal was to create a framework which balances the need to be flexible and adaptable with the need of simplicity and pragmatism to enable broad adoption [5].

OpenID Authentication provides a way to prove that an end user controls an Identifier. It does this without the RP needing access to end user credentials such as a password or to other sensitive information such as an email address. As mentioned already, OpenID is decentralized; so no central authority must approve or register RPs or OPs. An end user can freely choose which OpenID Provider to use, and can preserve their Identifier if they switch OpenID Providers. While nothing in the protocol requires JavaScript or modern browsers, the authentication scheme plays nicely with AJAX-style setups. This means an end user can prove their Identity to a Relying Party without having to leave their current Web page. OpenID Authentication uses only standard http(s) requests and responses, so it does not require any special capabilities of the User-Agent or other client software. OpenID is not tied to the use of cookies or any other specific mechanism of Relying Party or OpenID Provider session management. Extensions to User-Agents can simplify the end user interaction. OpenID Authentication is designed to provide a base service to enable portable, user-centric digital identity in a free and decentralized manner [1].

The authentication in OpenID framework happens at four different layers. *Identifier*, where the user is identified by a Address-based or Card-based identity; *discovery*, the users

associated identity services are discovered primarily by Yadis discovery protocol; *authentication*, multiple communication with the OP and RP to prove that the user owns a URL or i-name; *data transport*, which uses OpenID data transfer protocol for exchange of data between OP and RP [5]. The premise of OpenID is that, a user may assert an identity by using the own identifier. An OpenID relying party can then discover from that identifier the user's OpenID provider and initiate OpenID authentication [3].

2.1 Authentication in OpenID

Here there is a simple layman-explanation on how OpenID authentication works.

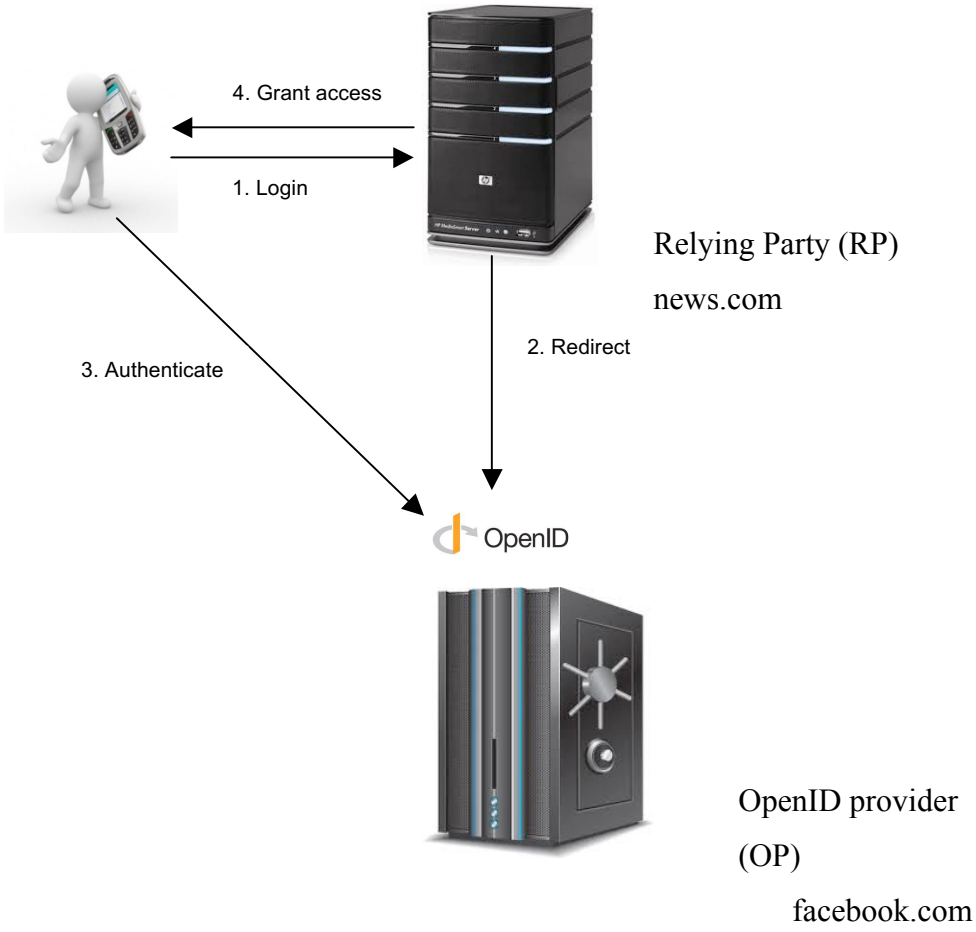


Figure 1 OpenID Authentication

The Figure 1 could be simply explained in the following manner. A client, who could be considered as an application, needs access to a service which potentially exists in one of the servers in a network. The server which hosts the service would ask the requester to prove the identity by taking to a login screen. The user could type in the credentials or could use an external OpenID provider which further requests the credentials. Upon successful authentication, the OpenID provider returns the authentication result to relaying party, the service provider.

2.2 OpenID in detail

The content in this section is an extract from OpenID 2.0 specification [1]. A step-by-step sequence of actions performed during OpenID authentication is explained here.

1. The end user initiates authentication by presenting a User-Supplied Identifier to the RP via their User-Agent.
2. After normalizing the User-Supplied Identifier, the RP performs discovery on it and establishes the OP Endpoint URL that the end user uses for authentication. The User-Supplied Identifier may be an OP Identifier, which allows selection of a Claimed Identifier at the OP or the protocol proceeds without a Claimed Identifier if that's being done via an extension.
3. The RP and the OP establish an association, a shared secret. The OP uses an association to sign subsequent messages and the RP to verify those messages; this removes the need for subsequent direct requests to verify the signature after each authentication request/response.
4. The RP redirects the end user's User-Agent to the OP with an OpenID Authentication request.
5. The OP establishes whether the end user is authorized to perform OpenID Authentication and wishes to do so.
6. The OP redirects the end user's User-Agent back to the RP with either an assertion that authentication is approved or a message that authentication failed.
7. The RP verifies the information received from the OP including checking the Return URL, verifying the discovered information, checking the nonce, and verifying the signature by using either the shared key established during the association or by sending a direct request to the OP.

2.2.1 OpenID Data Formats

This section describes the data formats which are supported in OpenID protocol. The OpenID recommends two types of data formats which are, protocol messages and integer representations. These are supported formats which are used for communication in OpenID.

a. Protocol Messages

The OpenID Authentication protocol messages are mappings of plain-text keys to plain-text values. The keys and values permit the full Unicode character set. When the keys and values

need to be converted to/from bytes, they are encoded using UTF-8. They cannot contain multiple parameters with the same name. They are further classified into Key-Value and HTTP encoding.

i. Key-Value Form Encoding

A message in Key-Value form is a sequence of lines. Each line begins with a key, followed by a colon, and the value associated with the key. The line is terminated by a single newline. A key or value does not contain a newline and a key also does not contain a colon. Additional characters, including whitespace, cannot be added before or after the colon or newline. Key-Value Form encoding is used for signature calculation and for direct responses to Relying Parties.

ii. HTTP Encoding

When a message is sent to an HTTP server, it is encoded using form-encoding. The keys in the request message are prefixed with *openid*. This prefix prevents interference with other parameters that are passed along with the OpenID Authentication message. When a message is sent as a POST, OpenID parameters are sent in, and extracted from, the POST body.
This model applies to messages from the User-Agent to both the RP and the OP, as well as messages from the RP to the OP.

b. Integer Representations

Arbitrary precision integers are encoded as big-endian signed two's complement binary strings. All integers that are used with Diffie-Hellman Key Exchange are positive. This means that the left-most bit of the two's complement representation is zero. If it is not, implementations add a zero byte at the front of the string.

2.2.2 OpenID Communication Types

OpenID communication type defines the ways in which the communication happens within the OpenID protocol. The communication types in OpenID are categorized as direct communication and indirect communication. The direct communication is initiated by a RP to an OP endpoint URL. The indirect communication are those passed through User-Agent and could be initiated either by RP or OP.

a. Direct Communication

The primary usage of direct communication is for establishing associations and verifying authentication assertions. A Direct Request is where the message is encoded as a POST body as HTTP encoding. All direct requests are HTTP POSTs. A Direct Response is the body of a response to a Direct Request consists of an HTTP Response body in Key-Value Form. This particular value is present for the response to be a valid OpenID 2.0 response. If this value is absent or set to *signon/1.0*, then this message is interpreted using OpenID Authentication 1.1 Compatibility mode. Upon success, a server receiving a valid request sends a response with an HTTP status code of 200. And for error responses, in which case, the response is malformed or contains invalid arguments, the server sends a response with a status code of 400.

b. Indirect Communication

Indirect communication is used for authentication requests and authentication responses. There are two methods for indirect communication: HTTP redirects and HTML form-submission. Both form-submission and redirection require that the sender know a recipient URL and that the recipient URL expect indirect messages. The initiator of the communication chooses which method of indirect communication is appropriate depending on capabilities, message size, or other external factors.

i. HTTP Redirect

Data can be transferred by issuing a 302, 303, or 307 HTTP Redirect to the end user's User-Agent. The redirect URL is the URL of the receiver with the OpenID Authentication message appended to the query string.

ii. HTML FORM Redirection

Indirect communication can also happen by a HTML Form redirection. This could be performed by mapping the keys to values and transferred by returning an HTML page to the User-Agent that contains an HTML form element. The form has a submit button.

iii. Indirect Error Responses

In the case of a malformed request, or one that contains invalid arguments, the OpenID Provider redirects the User-Agent to the *return_to* URL value if the value is present and it is a valid URL. The server could add additional keys to this response. If the malformed or invalid message is received by the Relying Party, *return_to* is not present or its value is not a valid URL, the server returns a response to the end user indicating the error and that it is unable to continue.

2.2.3 Initiation and Discovery

This section explains the identification process of Initiation, Normalization and Discovery in the Relying Party. This happens in the primary phase at the commencement of the authentication process in OpenID.

a. Initiation

OpenID Authentication is initiated by the Relying Party presenting the end user with a form that has a field for entering a User-Supplied Identifier. Browser extensions or other software that support OpenID Authentication may not detect a Relying Party's support if the *name* attribute is not set appropriately.

b. Normalization

The end user's input is normalized into an Identifier, as follows:

- If the user's input starts with the "xri://" prefix, it is stripped off, so that XRIs are used in the canonical form.
- If the first character of the resulting string is an XRI Global Context Symbol ("=", "@", "+", "$", "!"), then the input is treated as an XRI.
- Otherwise, the input is treated as an http URL. If it does not include an "http" or "https" scheme, the Identifier is prefixed with the string "http://". If the URL contains a fragment part, it is stripped off together with the fragment delimiter character "#".

URL Identifiers is further normalized by following redirects when retrieving their content and finally applying the rules to the final destination URL. This final URL is noted by the Relying Party as the Claimed Identifier and be used when requesting authentication.

c. Discovery

Discovery is the process where the Relying Party uses the Identifier to look up (discover) the necessary information for initiating requests. OpenID Authentication has three paths through which to do discovery:

- If the identifier is an XRI, it will yield an XRDS document that contains the necessary information. It should also be noted that Relying Parties can take advantage of XRI Proxy Resolvers. This will remove the need for the RPs to perform XRI Resolution locally.
- If it is a URL, the Yadis protocol is first attempted. If it succeeds, the result is again an XRDS document.
- If the Yadis protocol fails and no valid XRDS document is retrieved or no Service Elements are found in the XRDS document, the URL is retrieved and HTML-Based discovery is attempted.

Upon successful completion of discovery, the Relying Party will have one or more sets of the information. If more than one set of the information has been discovered, the precedence rules are applied. If XRI or Yadis discovery was used, the result will be an XRDS Document. This is an XML document with entries for services that are related to the Identifier. HTML-Based discovery is supported by Relying Parties. HTML-Based discovery is only usable for discovery of Claimed Identifiers. OP Identifiers are XRIs or URLs that support XRDS discovery.

HTML-Based discovery is used when an HTML document is available at the URL of the Claimed Identifier. The host of the HTML document could be different from the end user's OP's host.

2.2.4 Requesting Authentication

Once the RP has successfully performed discovery and (optionally) created an association with the discovered OP Endpoint URL, the RP sends an authentication request to the OP to obtain an assertion. This authentication request is an indirect request.

a. *Request Parameters*

openid.ns - Defines the namespace of the authentication request, value is "http://specs.openid.net/auth/2.0" otherwise, "http://openid.net/signon/1.1" or "http://openid.net/signon/1.0" for OpenID Authentication 1.1 Compatibility mode.

openid.mode - Defines the OpenID mode of the authentication request, value is "checkid_immediate" or "checkid_setup"

openid.claimed_id - Defines the Claimed Identifier (optional) of the authentication request.

openid.identity - Defines the OP-Local Identifier of the authentication request. Uses value of the claimed_id if not mentioned. Special value could be "http://specs.openid.net/auth/2.0/identifier_select" (optional).

openid.assoc_handle - Defines a handle for an association between the RP and the OP that is used to sign the response. If this is set, the transaction takes place in Stateless Mode (optional).

openid.return_to - Defines the URL to which the OP returns the User-Agent with the response indicating the status of the request. OP does not return to the end user if this is not set (optional).

openid.realm - Defines the URL pattern which the OP asks the end user to trust. This should be set if openid.return_to is omitted (optional).

b. Realms

A realm is a pattern that represents the part of URL-space for which an OpenID Authentication request is valid. A realm is designed to give the end user an indication of the scope of the authentication request. OPs present the realm when requesting the end user's approval for an authentication request. The realm is used by OPs to uniquely identify Relying Parties. For example, OPs can use the realm to allow the end user to automate approval of authentication requests. It is recommended that OPs protect their users from making assertions with overly-general realms. Overly general realms can be dangerous when the realm is used for identifying a particular Relying Party. Whether a realm is overly-general is at the discretion of the OP.

c. Immediate Requests

When requesting authentication, the Relying Party can request that the OP not interact with the end user. In this case the OP responds immediately with either an assertion that

authentication is successful, or a response indicating that the request cannot be completed without further user interaction.

2.2.5 Responding to Authentication Requests

When an authentication request comes from the User-Agent via indirect communication, the OP determines that an authorized end user wishes to complete the authentication. If an authorized end user wishes to complete the authentication, the OP sends a positive assertion to the Relying Party. If no association handle is specified, the OP uses a private association for signing the response. The OP stores this association and responds to later requests to check the signature of the response via Direct Verification. Relying Parties accepts and verify assertions about Identifiers for which they have not requested authentication. OPs use private associations for signing unsolicited positive assertions.

Positive assertions are indirect responses with some fields. If the OP is unable to identify the end user or the end user does not or cannot approve the authentication request, the OP sends a negative assertion to the Relying Party as an indirect response.

a. In Response to Immediate Requests

If the request was an immediate request, there is no chance for the end user to interact with pages on the OP to provide identifying credentials or approval of a request. In case of a negative assertion, the immediate request would return *openid.ns* and *openid.mode* as “setup_needed”.

b. In Response to Non-Immediate Requests

Since the OP may display pages to the end user and request credentials from the end user, a negative response to a request that is not immediate is definitive. Often, if the user does not wish to or cannot complete the authentication request, the OpenID authentication process will be aborted and the Relying Party will not get a cancel mode response (the end user may quit or press the back button in their User-Agent instead of continuing). If a RP receives the "cancel" response or authentication was unsuccessful, the RP treats the end user as non-authenticated.

2.2.6 Verifying Assertions

When the Relying Party receives a positive assertion, it verifies the following before accepting the assertion:

- The value of *openid.return_to* matches the URL of the current request.
- Discovered information matches the information in the assertion.
- An assertion has not yet been accepted from this OP with the same value for *openid.response_nonce*.
- The signature on the assertion is valid and all fields that are required to be signed are signed.

If all four of these conditions are met, assertion is now verified. If the assertion contained a Claimed Identifier, the user is now authenticated with that identifier.

2.2.7 Problems with OpenID

This section discusses some of the potential problems when using OpenID.

a. Lack of multilevel security

OpenID can save the overhead of the application site for authenticating the visiting user since the role of authentication can be played by the OpenID server. The user does not have to deal with many accounts since the resources of all websites are available for the OpenID user. By using OpenID, users' identity is unified, the operation processes are reduced and the system's security is enhanced. OpenID has provided an authentication platform which is URI based and extensible. As OpenID has become popularized in network applications, the security issue of OpenID authentication has also a topic of concern, e.g., the system information may be modified maliciously and a man in the middle could inject malicious code on the information system or create some other security hazards [34] [35].

b. Pharming and Phishing

Phishing is a way of attempting to acquire information (and sometimes, indirectly, money) such as usernames, passwords, and credit card details by masquerading as a trustworthy entity in an electronic communication. In most cases it involves stealing sensitive private information and finance account information by use of social engineering and technical concealment. Social engineering skill is to obtain sensitive private finance information such as credit card number, user ID, password etc from Internet, inducing users to their disguised

homepages by sending emails impersonating popular institution to many unspecified persons. Technical concealment is to obtain private information directly by installing malignant code such as Logger Spyware in private PC. Pharming, a fraud skill evolved from Phishing, steals private information by inducing users to disguised homepages by use of DNS hijacking. Users shall access correct site of finance authority induced by modulation of name decision system, but actually access fraudulent sites. Pharming is very high in its possibility to arouse damage to users while Phishing depending on social engineering [15].

c. Wrong approaches in Transport security

The endpoints of many OpenID Providers or Relying Parties are strictly HTTPS based. The problem is, if they are addressed via HTTP, they simply redirect the request to the HTTPS equivalent and proceed with the protocol flow. This section comprises the dangers of such a workaround. The User's Identifier is responsible for the OpenID Provider's endpoint. In general, the User is given his identifier by the OP, hence the OpenID Provider is overall responsible for the HTTP/HTTPS nature of its endpoint. Furthermore, the Relying Party sending an authentication request to the OpenID Provider is responsible for the return to parameter representing the Relying Party's endpoint, where the User will later be redirected to. If both of these endpoints are HTTP URLs, then both of the User's redirects are subject to forgery. The fact, that both of these parties may only allow communication over a TLS/SSL secured channel yields a false impression of security from the user's point of view.

d. Parameter Forgery

In this section, we exploit the message level security mechanism of OpenID - MAC. With respect to MACs, the two most important OpenID parameters are "*openid.sig*", representing the authentication code itself and "*openid.signed*" containing the hash value computed over all parameters XOR-ed with the pre-established shared key. The OpenID Authentication 2.0 protocol specification states, that if a positive assertion is received by the RP, it must not be accepted until it is verified. Any successful verification must satisfy, among others, the condition that 'the MAC of the assertion is valid and all required fields are MAC-protected'. Hence if a parameter is not defined as required and is not listed in "*openid.signed*", it is automatically subject to forgery. In other words, appending arbitrary unused parameters to a MAC-protected

message does not invalidate the assertion's MAC and the message stays intact and valid in the eyes of the Relying Party.

3. OAuth and Authorization

OAuth protocol enables websites or applications to access protected resources from a web service via an API, without requiring users to disclose their service provider credentials to the consumers. So it's a good candidate for secure web service call between service providers. OAuth authorization is the process in which users grant access to their protected resources without sharing their credentials with the consumer. OAuth uses tokens generated by the service provider instead of the user's credentials in accessing protected resources of the user. OAuth has been published as an open protocol so it's a good choice for the system to employ and to implement secure delegation access [21]. By authorizing the request token for the provider to grant the access token, the user can easily control access to owned resources, which is very essential in distributed systems involving 3rd parties.

3.1 OAuth Security

The first three properties discussed here are based on the authorization process and the last two on User service security.

Property P1: If a Consumer accesses User's data with an Access Token, then Service Provider must have issued the Access Token to this Consumer. *P1* is checked whether the transitions that represent Consumer accessing data are still reachable from the initial system state.

Property P2: If a Consumer obtains an Access Token by exchanging a Request Token, then the Request Token must have been authorized by User. *P2* is checked by removing the transitions of User authorization and check if the transitions for Consumer to exchange for Access Token are still reachable from the initial system state.

Property P3: If a Consumer obtains an Access Token by exchanging a Request Token, then the Service Provider must have issued the Request Token to this Consumer. *P3* is checked whether the transitions for Consumer to obtain Access Token are still reachable from the initial system state.

Property P4: If a Consumer accesses data that belong to User, then that User must have authorized the access. *P4* is checked by transitions for Consumer accessing User data are still reachable from the initial system state.

Property P5: If a Consumer accesses data that belong to a User, then only that User can get the service from Consumer with the accessed data – not any other User. *P5* is checked by authorizing the access or is redirected back to Consumer and transitions of Consumer accessing data are still reachable from the initial system state [22].

3.2 OAuth 2.0

The original OAuth1.0, initial release of the protocol, was primarily designed for web browsers and didn't provide profiles to support desktop and mobile devices. OAuth2.0 is the next evolution of the OAuth protocol, which greatly extend the client profiles by providing specific authorization flows for web browsers, desktop applications, and smart phones. OAuth2.0 introduced a long-lived refresh token that can be used to renew an access token with limited lifetime. In OAuth1.0, the lifetime of an access token is only set by the provider. The consumer has no way to renew the token after it is expired. So in a real application environment, the provider would like to issue a long lasting access token, typically valid for a year or unlimited lifetime to avoid repeated initiation of OAuth authorization flows for improving user experience. But such a practice could lead to potential security issues. OAuth2.0 introduces the renewal procedure that allows an OAuth consumer to hold a valid access token for a long time without bothering the user for the permission, and keep the access token limited under the control of the user [23].

3.3 Authorization with OAuth2.0

3.3.1 Simplified explanation – three legged dance

OAuth is a developer friendly technique providing specific authorization flows for web applications, desktop applications, mobile phones and living room devices.

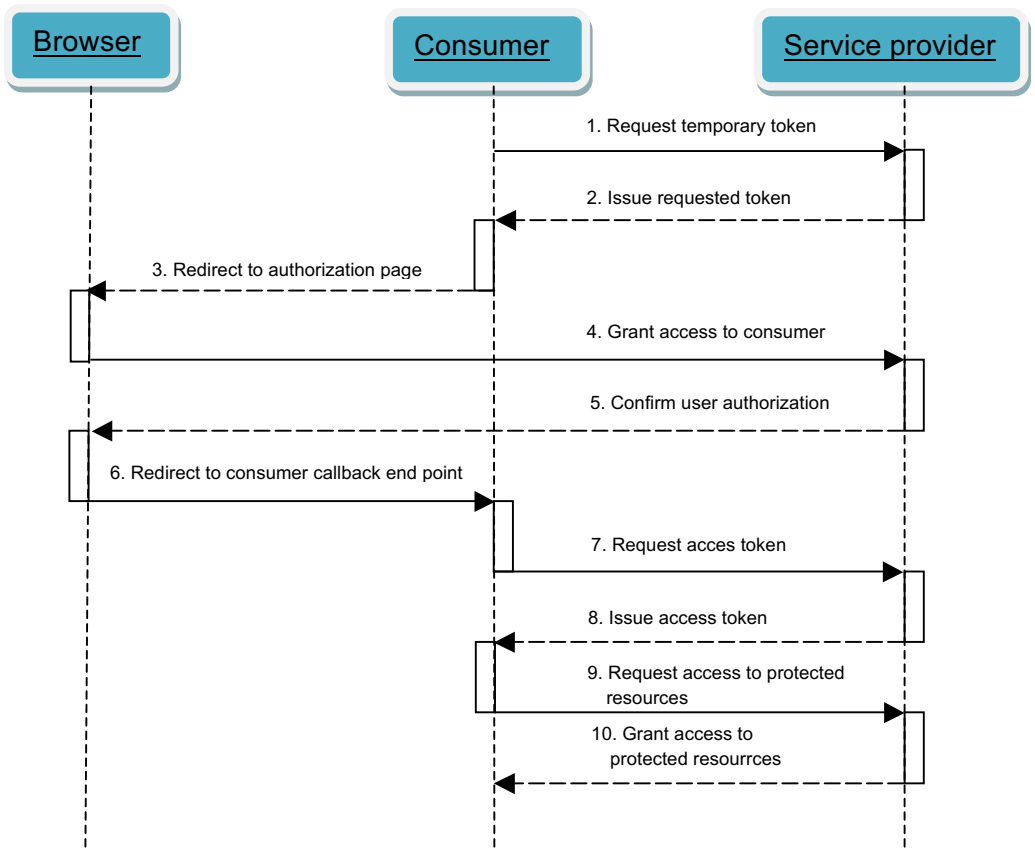


Figure 2 OAuth2.0 authorization

Step by step authorization with OAuth2.0 shown in Figure 2 is explained here:

1. The consumer requests a temporary token for the OAuth handshake. This token is used to maintain the handshake session.
2. After validating the consumer, the service provider issues a short-term request token.
3. The consumer sends an HTTP redirect response to the user's browser and leads the user to the service provider for authorization.
4. The user reviews the authorization request and grants access to the consumer on the service provider site if user trusts the consumer.
5. The service provider confirms the authorization and sends an HTTP redirect response to the user's browser.
6. The user's browser is redirected to the consumer's call-back URL, where the consumer completes the remaining part of the handshake.
7. The consumer requests the access token from the service provider with a verifier passed in the previous step.
8. Upon successful validation, the service provider issues the access token to access the protected resources.
9. After the OAuth handshake completes, the access token is issued and the consumer can use the access token to access the protected resources on behalf of the user.
10. The service provider validates each incoming OAuth request and returns the protected resources if the consumer is authorized.

3.4 OAuth in detail

The following is an extract from OAuth2.0 specification [3]. This describes the OAuth implementation in context with the explanation above.

3.4.1 Registration

Before initiating the protocol, the client registers with the authorization server. This typically involves end-user interaction with an HTML registration form. Client registration does not require a direct interaction between the client and the authorization server. If supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g. redirection URI, client type). OAuth2.0 differentiates the client as confidential or public based on authenticating securely with authorization server. Interaction with OAuth could happen as a user-agent-based application, native application or as a web application. There is always a client identifier associated with the registration. Confidential clients are typically issued a set of client credentials used for authenticating with the authorization server (e.g. password, public/private key pair).

3.4.2 Endpoints

OAuth2.0 authorization process utilizes two authorization server endpoints based on http resources. Authorization endpoint is used by the client to obtain authorization from the resource owner via user-agent redirection. The token endpoint is used by the client to exchange an authorization grant for an access token, typically with client authentication. There is also Redirection endpoint used by the authorization server to return authorization credentials responses to the client via the resource owner user-agent.

i. Authorization endpoint

The endpoint URI includes an "application/x-www-form-urlencoded" formatted query component, which is retained when adding additional query parameters. The endpoint URI does not include a fragment component. Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server uses TLS when sending requests to the authorization endpoint.

ii. Token endpoint

The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly). The endpoint URI includes an "application/x-www-form-urlencoded" formatted query component which is retained when adding additional query parameters. The endpoint URI does not include a fragment component. Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server requires TLS when sending requests to the token endpoint. Also the client uses the HTTP "POST" method when making access token requests. Those parameters sent without a value are treated as if they were omitted from the request. The authorization server ignores unrecognized request parameters. Request and response parameters are not included more than once.

3.4.3 Obtaining Authorization

To request an access token, the client should obtain authorization from the resource owner. The authorization is expressed in the form of an authorization grant which the client uses to request the access token. OAuth defines four grant types: *authorization code, implicit, resource owner password credentials and client credentials*. It also provides an extension mechanism for defining additional grant types.

i. Authorization code grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. As a redirection-based flow, the client is capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

The client constructs the *authorization request* URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format. The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent. The authorization server upon validation of the request authenticates the resource owner and obtains an authorization decision. When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

After granting the access request from resource owner, the authorization server issues an authorization code and delivers it to the client by adding parameter to the query component of the redirection URI using the "application/x-www-form-urlencoded" format as *authorization response* If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server informs the resource owner of the error and does not automatically redirect the user-agent to the invalid redirection URI. If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, it is considered as an *error response*. Then the authorization server informs the client by adding parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded"

The client makes an *access token request* to the token endpoint by adding parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body. If the access token request is valid and authorized, the authorization server issues an access token as an *access token response* and optional refresh token. If the request client authentication failed or is invalid, the authorization server returns an error response.

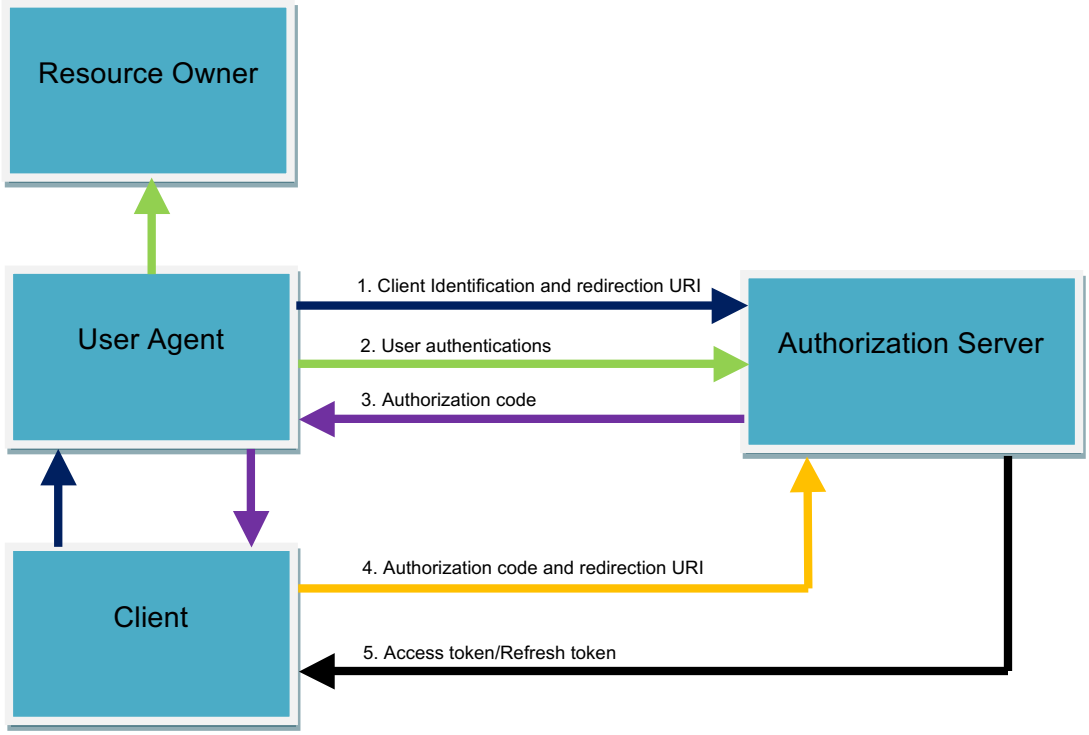


Figure 3 OAuth2.0 authorization code grant

The Figure 3 illustrates how an authorization code grant takes place. This happens in five steps which are explained below.

(1) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI for the authorization server to send the user-agent back once access is granted (or denied).

(2) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(3) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

(4) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

(5) The authorization server authenticates the client, validates the authorization code, and ensures the redirection URI received matches the URI used to redirect the client in step (3). If valid, the authorization server responds back with an access token and optionally, a refresh token.

ii. Implicit grant

The implicit grant type is used to obtain access tokens (it does not support the issue of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are implemented in a browser using a scripting language such as JavaScript. As a redirection-based flow, the client is capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server. Unlike the authorization code grant type in which the client makes separate requests for authorization and access token, the client receives the access token as the result of the authorization request. The implicit grant type does not include client authentication and relies

on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device.

The *authorization request*, *access token response* and *error response* are exactly the same as the ones discussed in the *authorization code grant* section.

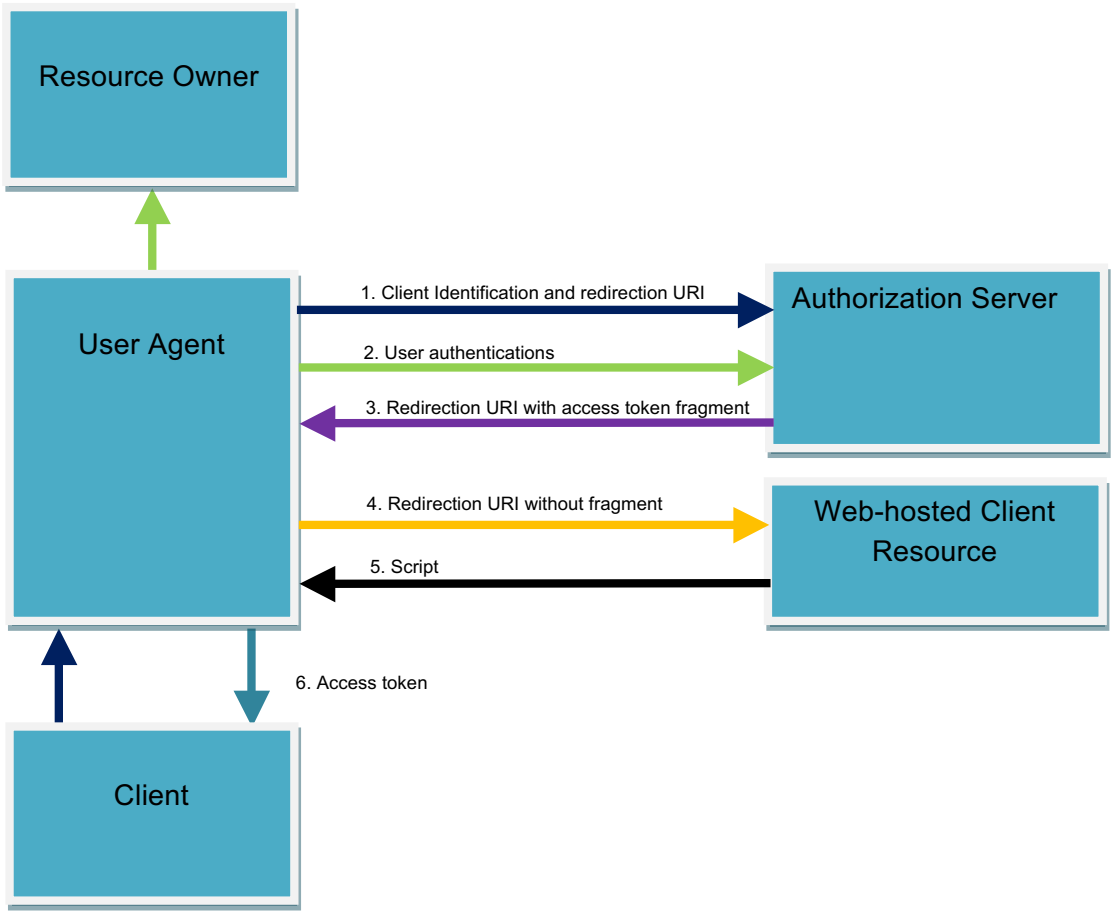


Figure 3 OAuth2.0 authorization implicit grants

The Figure 4 illustrates how an authorization implicit grant takes place. This happens in six steps which are explained below.

(1) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(2) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(3) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.

(4) The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment). The user-agent retains the fragment information locally.

(5) The web-hosted client resource returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.

(6) The user-agent executes the script provided by the web-hosted client resource locally, which extracts the access token and passes it to the client.

iii. Resource owner password credentials

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. The authorization server takes special care when enabling this grant type, and only allows it when other flows are not viable. This grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password) typically using an interactive form. It is also used for migration of existing clients using direct authentication schemes to OAuth2.0 by converting the stored credentials to an access token.

The client makes an *access token request* to the token endpoint by adding parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body. If the client type is confidential or the client was issued the client credentials, the client authenticates with the authorization server. Since this access token request utilizes the resource owner's password, the authorization server protects the endpoint against brute force attacks like using rate-limitation or generating alerts.

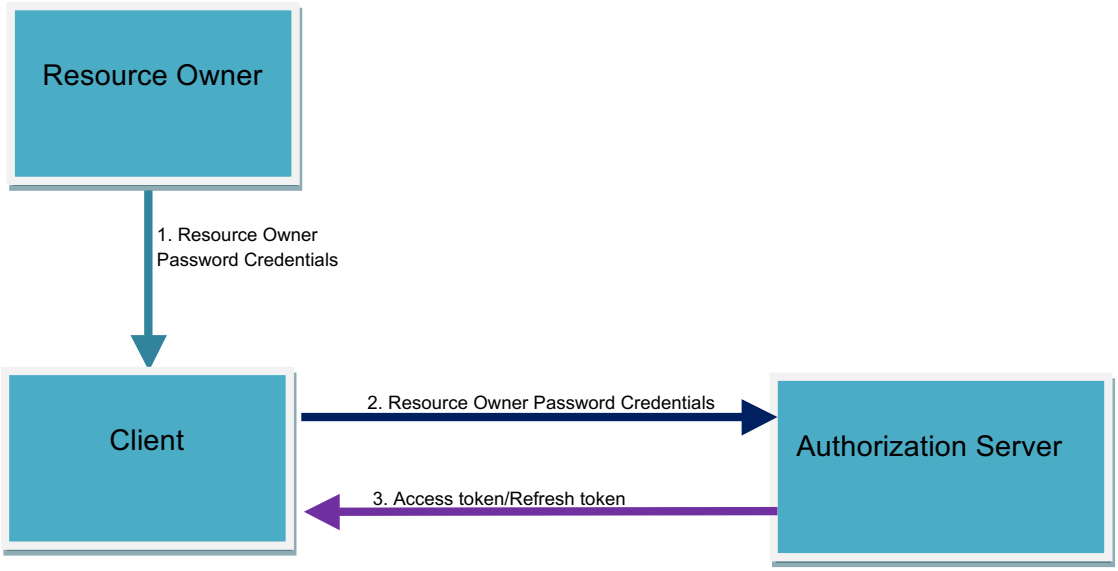


Figure 3 OAuth2.0 authorization resource owner password credentials

The Figure 5 illustrates how a resource owner password credential authorization takes place. This happens in three steps which are explained below.

(1) The resource owner provides the client with its username and password.

(2) The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.

(3) The authorization server authenticates the client and validates the resource owner credentials, and if valid issues an access token.

iv. Client credentials grant

The client can request an access token using only its client credentials when the client is requesting access to the protected resources under its control or those of another resource owner which has been previously arranged with the authorization server.

The *authorization request*, *access token request/response* and *error response* are exactly the same as the ones discussed in the *authorization code grant* section.

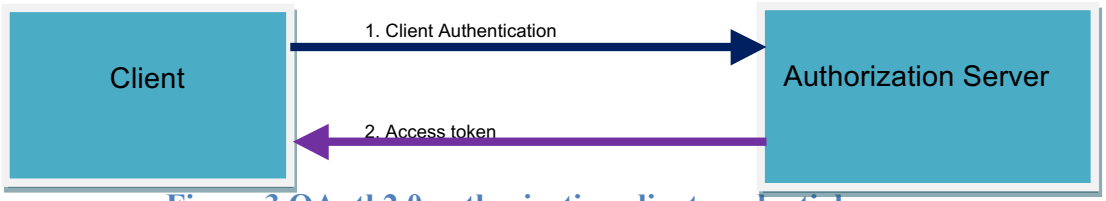


Figure 3 OAuth2.0 authorization client credentials

The Figure 6 illustrates how an authorization client credentials takes place. This happens in two steps which are explained below.

(1) The client authenticates with the authorization server and requests an access token from the token endpoint.

(2) The authorization server authenticates the client. If valid, then issues the access token.

v. Extension grants

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the "*grant_type*" parameter of the token endpoint, and by adding any additional parameters necessary. If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token. If the client authentication request fails or is invalid, the authorization server returns an error response.

3.4.4 Issuing and refreshing an access token

This section describes the specification of a successful response and an error response. There is also discussed the responses in refreshing an access token.

a. Successful Response

The authorization server issues an access token and optional refresh token and constructs the response by adding parameters to the entity body of the HTTP response with a 200 (OK) status code. It is to be noted that the parameters are included in the entity body of the HTTP response using the "*application/json*" media type. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter in these JSON responses. The authorization server includes the HTTP "*Cache-Control*" response header field with a value of "*no-store*" in the response containing tokens, credentials, or other sensitive information, as well as the "*Pragma*" response header field with a value of "*no-cache*". The client ignores unrecognized value names in the response.

b. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code and includes parameters with the response. Similar to the case of successful response, the parameters are included in the entity body of the HTTP response using the "*application/json*" media type. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter in these JSON responses.

c. Refreshing an Access Token

If the authorization server issues a refresh token to the client, the client makes a refresh request to the token endpoint by adding the parameters using the *"application/x-www-form-urlencoded*" format in the HTTP request entity-body. Because refresh tokens are typically long-lasting credentials used to request additional access tokens, the refresh token is bound to the client to whom it was issued. If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client authenticates with the authorization server. For example, the client makes the HTTP request using transport-layer security. If valid and authorized, the authorization server issues an access token. If the request fails the verification or is invalid, the authorization server returns an error response. The

authorization server issues a new refresh token, in which case the client discards the old refresh token and replace it with the new refresh token. The authorization server also revokes the old refresh token after issuing a new refresh token to the client. With the new refresh token issued, the refresh token scope is identical to that of the refresh token included by the client in the request.

3.4.5 Accessing protected resources

The client accesses the protected resources by presenting the access token to the resource server. The resource server validates the access token, ensures that it has not expired and has a scope for the requested resource. The methods used by the resource server to validate the access token generally involve an interaction or coordination between the resource server and the authorization server. The method in which the client utilizes the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP "*Authorization*" request header field with an authentication scheme defined by the access token type specification.

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client does not use an access token if it does not understand the token type.

Some examples of access token types are:

a. *The "bearer" token type*

   This is utilized by simply including the access token string in the request

b. *The "mac" token type*

This is utilized by issuing a MAC key together with the access token which is used to sign certain components of the HTTP requests.

Each access token type definition specifies the additional attributes (if any) sent to the client together with the "*access_token*" response parameter. It also defines the HTTP authentication method used to include the access token when making a protected resource request.

3.4.6 Extensibility

This section describes the specification for extending custom-made parameters, tokens and grant types. Here is discussed, defining new access token types, endpoint parameters, authorization grant types, authorization endpoint response types and additional error codes.

a. Defining new access token types

Access token types can be defined in two ways, either by registering in the access token type registry or by using a unique absolute URI as its name. Types utilizing a URI name are limited to vendor-specific implementations that are not commonly applicable and are specific to the implementation details of the resource server where they are used. All other types are registered. If the type definition includes a new HTTP authentication scheme, the type name is identical to the HTTP authentication scheme name.

b. Defining new endpoint parameters

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the parameters registry. Unregistered vendor-specific parameter extensions that are not commonly applicable and are specific to the implementation details of the authorization server where they are used utilize a vendor-specific prefix that does not conflict with other registered values (e.g. beginning with 'companyname_').

c. Defining new authorization grant types

New authorization grant types can be defined by assigning them a unique absolute URI for use with the "*grant_type*" parameter. If the extension grant type requires additional token endpoint parameters, they are registered in the OAuth parameters registry

d. Defining new authorization endpoint response types

New response types for use with the authorization endpoint are defined and registered in the authorization endpoint response type registry. If a response type contains one or more space characters (%x20), it is compared as a space-delimited list of values in which the order of values does not matter. Only one order of values is registered, which covers all other arrangements of the same set of values. For example, the response type "token code" is left undefined by Auth2.0

specification. However, an extension can define and register the "*token code*" response type. Once registered, the same combination cannot be registered as "*code token*", but both values can be used to denote the same response type.

e. Defining additional error codes

In cases where protocol extensions (i.e. access token types, extension parameters, or extension grant types) additional error codes to be used with the authorization code grant error response, the implicit grant error response or the token error response, such error codes are defined. Extension error codes are registered if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type.

4. OpenID and OAuth combination

The emerging authentication and delegation technologies of OpenID and OAuth are gathering much attention. In these technologies, components are allowed to establish sessions among more than two components. By verifying a trust chain among them, new sessions are established without prior sharing of credentials (e.g. a password). OpenID provides the single sign-on feature; users who have been authenticated by the authentication server, can establish sessions with other servers. OAuth allows users to grant their access authorities to servers, which use the granted authorities when establishing new sessions with other servers. Several large Web sites including Google and Yahoo have already introduced these technologies, because they are essential for permitting modern Web sites to interwork with each other by flexibly establishing sessions [24].

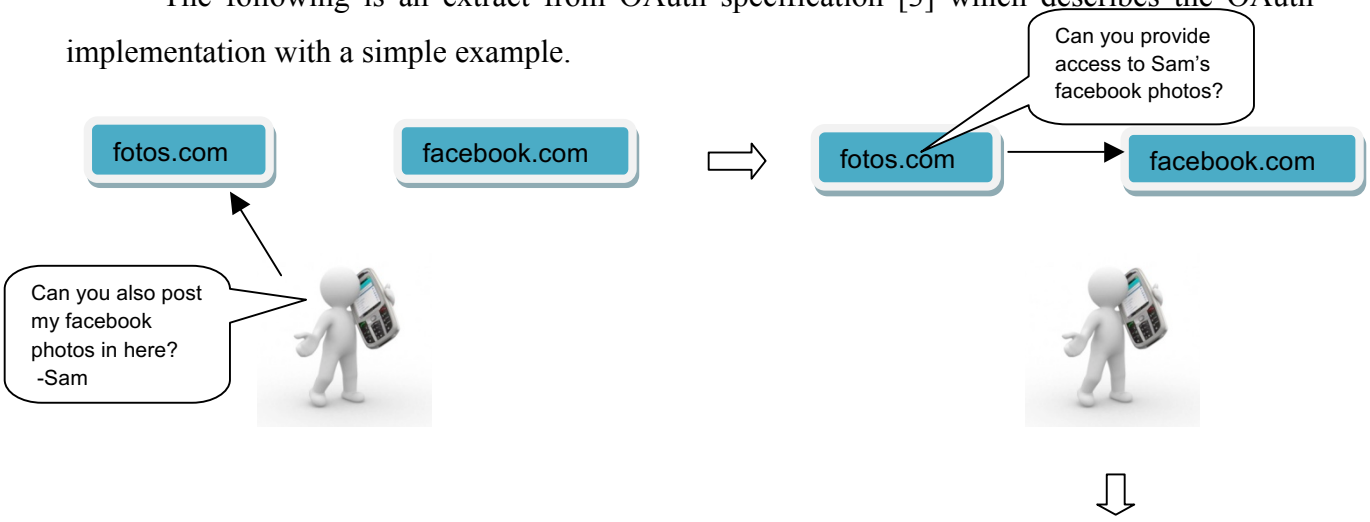
A hybrid approach that combines OpenID and OAuth together is called as OpenID OAuth Extension. It describes a mechanism to combine an OpenID authentication request with the approval of an OAuth request token. After the user is authenticated on the OpenID provider, the provider asks for the user's approval for relying party's (RP) unauthorized token for this user. After the user is redirected back to RP, the RP can exchange the approved request token to an access token that can further consume the OP's services on behalf of the current user [21].

The OpenID OAuth Extension describes the integration of OpenID Authentication and OAuth Core specifications together. It is assumed that the OpenID Provider and OAuth Service Provider are the same service. The extension insists on a combined authentication and authorization screen for the two protocols in order to provide good user experience. Also this extension recommends embedding an OAuth approval request into an OpenID authentication request to permit combined user approval. For security reasons, the OAuth access token is not returned in the OpenID authentication response. Instead a mechanism to obtain the access token is provided. The OpenID OAuth Extension does not provision request tokens in a server-to-server request from the Combined Consumer (CC) to the request token endpoint at the Combined Provider (CP). Instead, the CP returns an already-approved request token to the CC as part of the OpenID authentication response. The CC then exchanges the request token for an access token at the access token endpoint of the CP, following standard OAuth practice. Before requesting

authentication registration, the CC and the CP agree on a consumer key and consumer secret. The CP also obtains a list of valid OpenID realms that the CC may use in subsequent authentication requests from CC well before. After which, the CP verifies that CC is authorized to use those realms. While requesting OpenID Authentication via the protocol mode "*checkid_setup*" or "*checkid_immediate*", this extension can be used to request that the end user authorize an OAuth access token at the same time as an OpenID authentication. This is done by sending the following parameters as part of the OpenID request. Authorizing the OAuth request is performed by checking if the OpenID OAuth Extension is present in the authentication request and by verifying the authorization of the consumer key passed in the request by CP. If the verification succeeds, the CP determines that delegation of access from a user to the CC has been requested. The CP does not issue an approved request token unless it has user consent to perform such delegation. If the OpenID authentication request was not fulfilled, then the OAuth request is considered to fail and the CP does not send any OpenID OAuth Extension values in the response. The OAuth Authorization was declined or not valid, the CP only responds with the parameter "*openid.ns.oauth*". The Access Token is obtained by exchanging the request token for an access token. The CP sends an access token request to the access token endpoint of the CP. The CP verifies the request and either issue the access token or sends an error response [33].

4.1 Example case with OpenID+OAuth

The following is an extract from OAuth specification [3] which describes the OAuth implementation with a simple example.



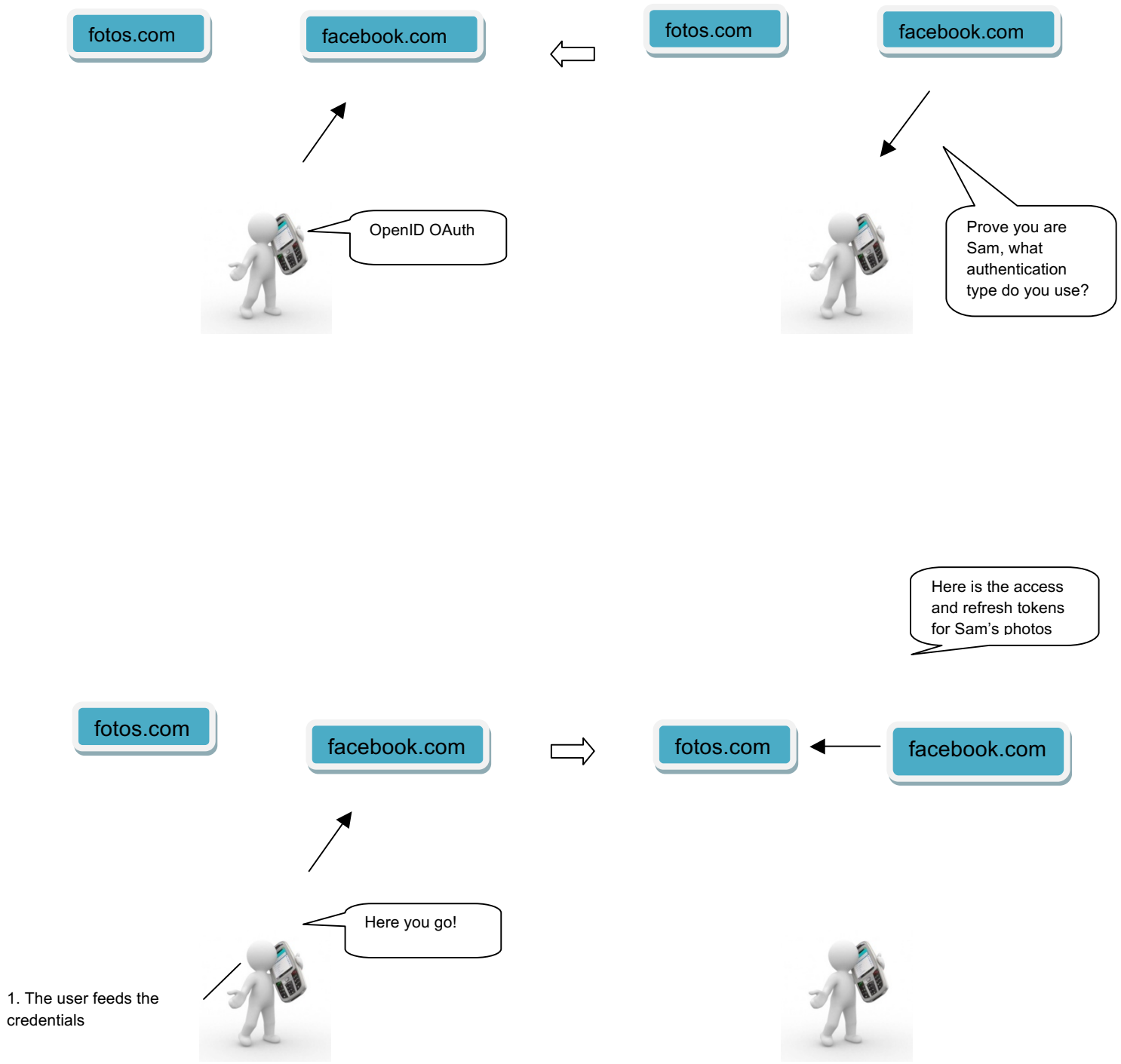


Figure 3 Combined OpenID+OAuth example usage

Figure 3 describes a simple example work flow of authentication process with OAuth2.0. It's a typical authorization scenario, where a 3rd party wants to access a user's (Sam in this case) protected resource existing in another server. This example is similar to the one given in the OAuth2.0 specification.

The User Sam wants fotos.com to display the pictures uploaded in facebook.com in fotos.com immediately or at regular intervals. It could be that, fotos.com advertises users like Sam that they could provide one-stop to display all the pictures and videos from various websites where Sam has account and has been uploading pictures and videos. After user grants permission to this action, fotos.com redirects the user to facebook.com which further redirects user to a prompt where Sam has to prove the identity. However, Sam is given options as to choose which way to prove it. Sam chooses OpenID, then Sam is redirected again to a prompt where Sam gets a confirmation message to use the existing one unless Sam hasn't created one yet. Otherwise Sam has an option of hand-typing the credentials, i.e. user-id and password to the browser. The password is sent over the Internet. Upon successful authentication, Sam is asked to authorise the request by fotos.com and by further accepting, the access token and refresh tokens are granted to fotos.com to access Sam's photos in facebook.com.

The primary problem cause occurs when Sam types the credentials and passes it over Internet. One of the biggest problems because of such an action of Sam is Internet phishing. This is discussed in the

Problem **Scenarios** section.

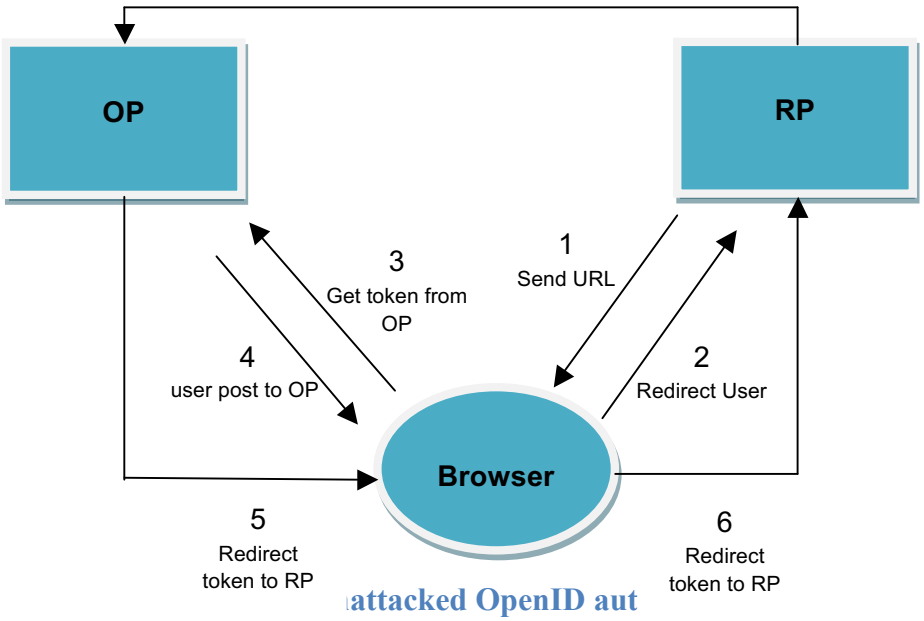
5. Problem Scenarios

5.1 Scenario 1: Internet phishing in OpenID

There have been several studies performed on phishing aspects of OpenID. Here one of the scenarios is discussed.

Considering the example case from the previous section of the document, there is nowhere the client's authenticity is checked in OpenID authentication process. In other words, it's impossible to check the authenticity of each and every client which exists in the whole Internet world. Think of a situation when user triggers the same action with a malicious client. The client could redirect the user to a bogus host which could resemble the look and feel of real host. For the OpenID's case, it's also possible to manipulate the OpenID server like host, which resembles the real OP. The user, irrespective of using OpenID ends up providing the valuable credentials in wrong hands. In case of OpenID credentials, it's much more critical information, as they allow access to multiple service providers.

An extract from [25] is being discussed here to obtain of clear understanding of Internet phishing attacks with OpenID. A picture of registration back channel and an attacked OpenID authentication via a malicious RP is shown in the Figure 4 and 5 respectively.



attacked OpenID aut

From the figure 4, an interaction starts with the RP telling the user what the URL is (1). Then the procedure redirects the user to the OP to pick up an authentication token (2) and (3). To perform the authentication, the OP has to be sure that it's the user who is making the request. So it presents with an authentication screen, typically asking for a username and password (4). If they are entered correctly, the OP mints a token to send to the RP as shown in (5) and (6). If the OP and RP already know each other, this is the end of the authentication part of the protocol.

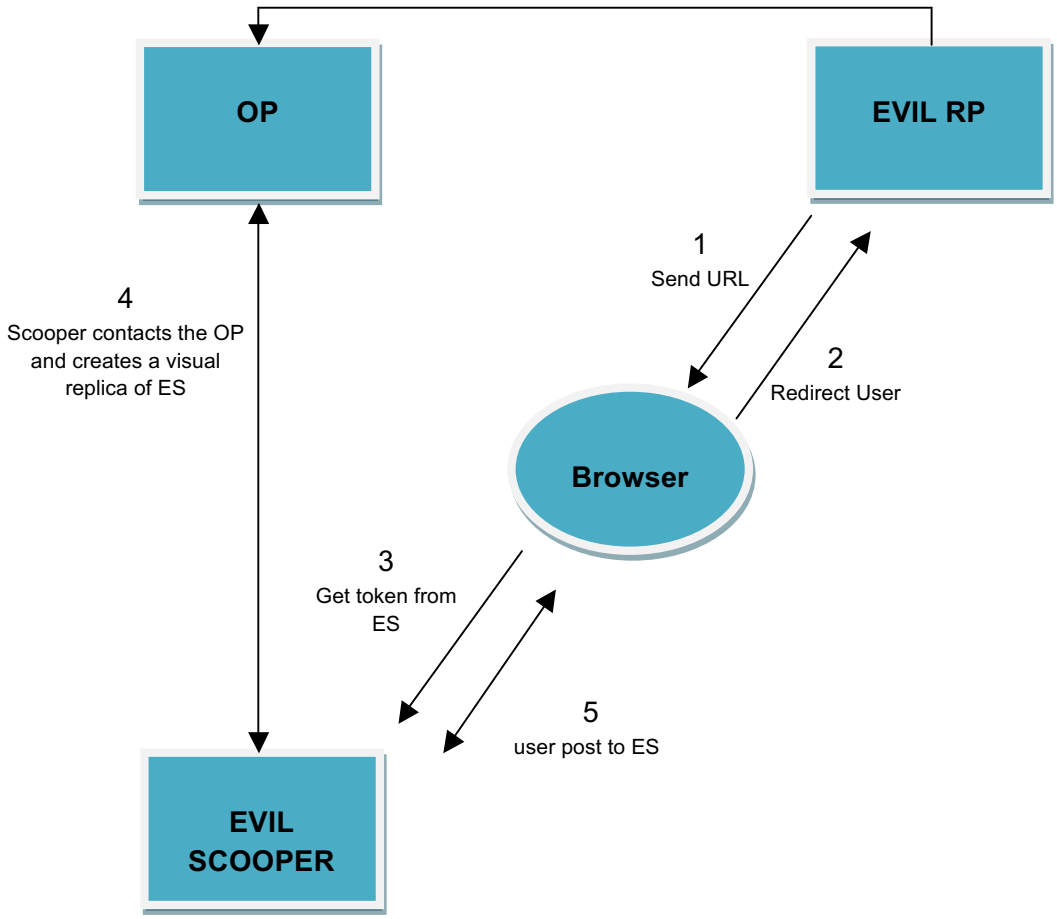


Figure 5 Attacked OpenID authentication

The figure 5 shows one of the ways how OpenID authenticated system could be a victim of Internet phishing. The user unwittingly goes to an evil site through conventional phishing or by following a search engine. The user is sent the evil RP URL (1). But instead of redirecting the user to the legitimate OP, it redirects to the Evil Scooper site (2) and (3). The Evil Scooper contacts the legitimate OP and pulls down an exact replica of its login experience ("man in the middle") (4). The user posts the credentials (username and password) which can now be used by the Evil Scooper to get tokens from the legitimate OP. These tokens can then be used to gain access to any legitimate RP.

5.2 Scenario 2: OpenID and OAuth complexity

Setting up an OpenID with OAuth server would involve steps like designing a login User Interface, selecting an OpenID compliant library, creating the mechanism for performing discovery and making authentication requests, adding OAuth capability to authentication requests, create a mechanism to extract and store the information returned by RP. Though the protocol provides smooth handovers, every new user per user terminal should undergo all the steps mentioned above just to get logged into a site. This could be far simplified if these authentication and authorization solutions are provided by a single protocol itself.

6. Kerberos and authentication

Kerberos V5 is an authentication system developed at MIT. Kerberos is named for the three-headed watchdog from Greek mythology, which guarded the entrance to the underworld [32]. Kerberos is a network authentication protocol communicating over a non-secure network to prove their identity to one another in a secure manner. This is very reliable and has a proven security analysis for authenticated encryption [26], [27].

It makes use of a trusted third party Key Distribution Center (KDC), which consists of two independent roles: an Authentication Server (AS) and a Ticket Granting Server (TGS) as shown in Figure 7. Under Kerberos, a client (either a user or a service) sends a request for a ticket to the Key Distribution Center (KDC). The KDC creates a ticket-granting ticket (TGT) for the client, encrypts it using the client's password as the key, and sends the encrypted TGT back to the client. The client then attempts to decrypt the TGT, using its password. If the client successfully decrypts the TGT (i.e., if the client gave the correct password), it keeps the decrypted TGT, which indicates proof of the client's identity. The TGT, which expires at a specified time, permits the client to obtain additional tickets, which give permission for specific services. The requesting and granting of these additional tickets is user-transparent. Since Kerberos negotiations are authenticated and optionally encrypted, communications between two points anywhere on the Internet provides a layer of security that is not dependent on which side of a firewall either client is on. Kerberos is a single-sign-on system, which means that you have to type your password only once per session, and Kerberos does the authenticating and encrypting transparently [32].

6.1 Kerberos in detail

This section is an extract from Kerberos specification [32].

6.1.1 Kerberos Ticket

The Kerberos credentials, or tickets, are a set of electronic information that can be used to verify your identity. Your Kerberos tickets may be stored in a file or they could exist only in

memory. The first ticket obtained is a ticket-granting ticket, which permits to obtain additional tickets. These additional tickets give permission for specific services. The requesting and granting of these additional tickets happens transparently.

A good analogy for the ticket-granting ticket is a three-day ski pass that could be used at four different resorts. The pass is shown at whichever resort one decides to go to (until it expires) and receives a lift ticket for that resort. Once the lift ticket is gotten, one can ski all long at that resort. Upon arriving another resort, the next day, pass is shown once again and an additional lift ticket for the new resort is gotten. The difference is that the Kerberos V5 programs notice that you have the weekend ski pass and get the lift ticket, so the transactions are performed itself.

6.1.2 Kerberos principal

Kerberos Principal is a unique identity to which Kerberos can assign tickets. Principals can have an arbitrary number of components. Each component is separated by a component separator, generally "/". The last component is the realm, separated from the rest of the principal by the realm separator, generally "@". If there is no realm component in the principal, then it will be assumed that the principal is in the default realm for the context in which it is being used.

Traditionally, a principal is divided into three parts: the primary, the instance and the realm. The format of a typical Kerberos V5 principal is primary/instance@REALM.

- The primary is the first part of the principal. In the case of a user, it's the same as username. For a host, the primary is the word host. The instance is an optional string that qualifies the primary.
- The instance is separated from the primary by a slash (/). In the case of a user, the instance is usually empty. A user might also have an additional principal, with an instance called 'admin', which the person uses to administrate a database. The principal jennifer@ATHENA.MIT.EDU is completely separate from the principal jennifer/admin@ATHENA.MIT.EDU, with a separate password, and separate permissions. In the case of a host, the instance is the fully qualified hostname, e.g., daffodil.mit.edu.
- The realm is the Kerberos realm. In most cases, your Kerberos realm is your domain name, in upper-case letters. For example, the machine daffodil.example.com would be in the realm EXAMPLE.COM.

6.1.3 Kerberos Ticket management

On many systems, Kerberos is built into the login program and tickets are automatically gotten when you log in. Some programs can forward copies of user's tickets to the remote host. Most of those programs also automatically destroy the tickets when they exit. However, MIT recommends that it's good to explicitly destroy the Kerberos tickets when it's through with them, just to make sure. Additionally, it is safest to either destroy all copies of the tickets.

a. Kerberos Ticket Properties

Brief explanations of the different types of tickets are discussed here.

Ticket forwarding: If a ticket is forwardable, then the KDC can issue a new ticket with a different network address based on the forwardable ticket. This allows for authentication forwarding without requiring a password to be typed in again. For example, if a user with a forwardable TGT logs into a remote system, the KDC could issue a new TGT for that user with the network address of the remote system, allowing authentication on that host to work as though the user were logged in locally. When the KDC creates a new ticket based on a forwardable ticket, it sets the forwarded flag on that new ticket. Any tickets that are created based on a ticket with the forwarded flag set will also have their forwarded flags set.

Proxiable ticket: It is similar to a forwardable ticket that allows a service to take on the identity of the client. Unlike a forwardable ticket, a proxiable ticket is only issued for specific services. In other words, a TGT can be issued based on a ticket that is forwardable but not proxiable.

Postdated ticket: It is issued with the invalid flag set. After the starting time listed on the ticket, it can be presented to the KDC to obtain valid tickets. Tickets with the postdateable flag set can be used to issue postdated tickets.

Renewable tickets: This can be used to obtain new session keys without the user entering their password again. A renewable ticket has two expiration times. The first is the time at which that particular ticket expires. The second is the latest possible expiration time for any ticket issued based on this renewable ticket.

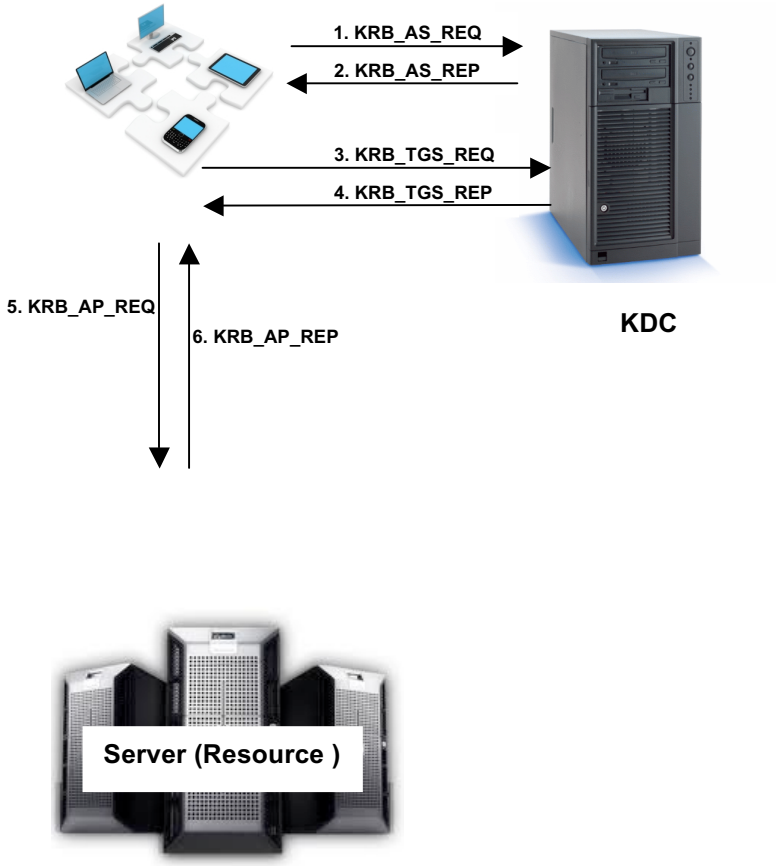
Invalid ticket: It is a ticket which is rejected by application servers. Postdated tickets are usually issued with this flag set and is validated by the KDC before they can be used.

Preauthenticated ticket: It is only issued after the client requesting the ticket had authenticated itself to the KDC. The hardware authentication flag is set on a ticket which required the use of hardware for authentication. The hardware is expected to be possessed only by the client who requested the tickets.

Anonymous ticket: It is the one in which the named principal is a generic principal for that realm. It does not actually specify the individual that will be using the ticket. This ticket is meant only to securely distribute a session key.

6.2 Kerberos negotiations

Here is a simple explanation on how Kerberos negotiations happen during an authentication process. There exists a client who needs to authenticate the server. The Key Distribution Centre shown as a separate entity could exist within the server or external to the server.



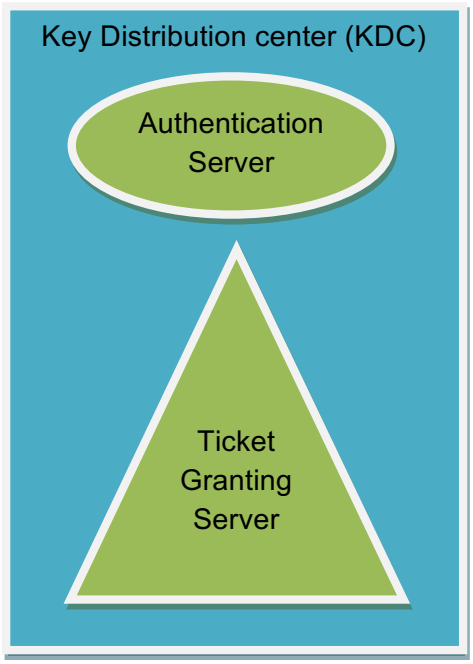


Figure 6 Kerberos negotiations

Figure 7 KDC block

1. The client sends a KRB_AS_REQ to the KDC and more specifically the Authentication Server to request a Ticket Granting Ticket (TGT).
2. Once the KDC verifies the users Authentication Data, it responds back to the client with a KRB_AS_REP to the client with a TGT and session.

3. The client is then able to request service tickets since it has a valid TGT. The client then sends a KRB_TGS_REQ to the Ticket Granting Server to request a Service Ticket.
4. Once the KDC has verified the validity of the TGT that is included with the Service Ticket request, it responds back to the client with a KRB_TGS_REP with the Service Ticket and service session key.
5. Next the client sends the Service Ticket to the Service/Application as a KRB_AP_REQ.
6. After authentication succeeds the Service responds back to the client with a KRB_AP_REP.

6.3 Kerberos limitations

The password is the only way Kerberos has as a verifying identity. If someone finds out the password, that person can masquerade as user, send email that comes from user, read, edit, or delete files or log into other hosts as user. No one will be able to tell the difference. For this reason, it is important that one should choose a good password and keep it secret. If user needs to give access to the account to someone else, user can do so through Kerberos by Granting Access to the Account. There is no need for the user to tell your password to anyone, including the system administrator, for any reason. User should change the password frequently [32].

Conventional Kerberos does not operate as an open system because every user must be known a priori. A shared secret between the AS and the user (a password-derived key) must be maintained by the AS and each user has a one-to- one mapping with a principal name. Most Kerberos extensions are not designed to make Kerberos operate as an open system. PKINIT and other public-key extensions extend credential management to third parties (trusted CAs), but the third parties usually cooperate directly with the Kerberos administrator in creating certificates with principal names that exist in the database [28].

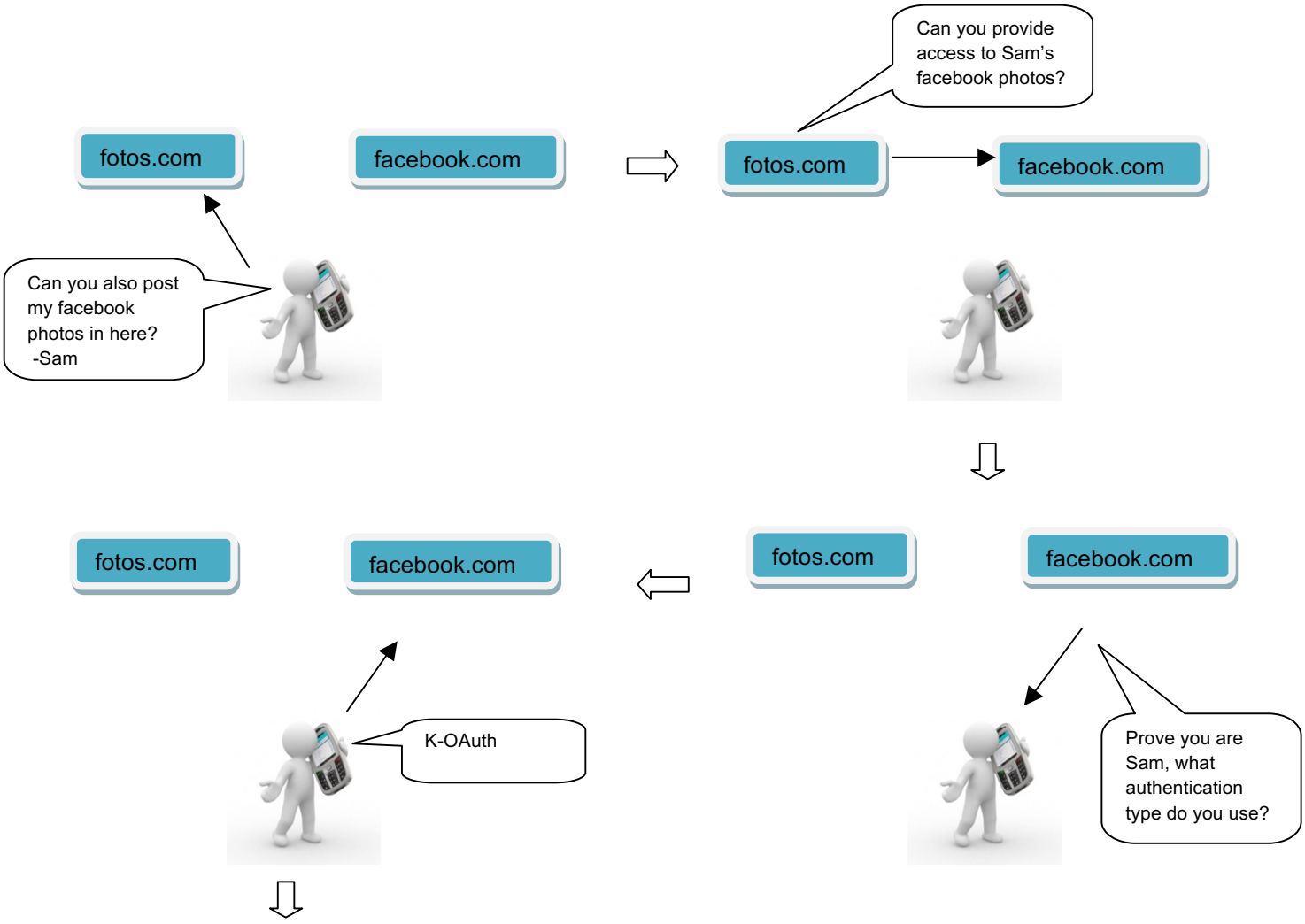
7. Evolution of K-OAuth (Kerberos OAuth)

In a traditional OpenID-OAuth transaction, when a user wants to authorize a 3rd party to access to the protected resources of the user in another domain, the domain which contains the user's protected resource prompts for identity verification of the user. The user could provide the hand-typed credentials or opt for OpenID, which might send the credentials over TCP/IP. This is the crust of the problem. When the credentials of the user are sent across the network, especially through internet, there are huge vulnerabilities of the credentials getting stolen via internet phishing etc. The Kerberos-OAuth, implements Kerberos for authentication uses the password of the user locally and never transmits them over internet. Instead, Kerberos keys which are encrypted are transferred to the authentication server(s) for authentication. The K-OAuth combines the authentication and authorization into single unit to prevent most of the problems occurred due to credentials sent over networks.

The OAuth2.0 version of OAuth uses token exchange mechanism for authorizing which 3rd party entities use to obtain access to resources residing in another domain. By means of extending OAuth the Kerberos way, authentication could also be performed with OAuth protocol. In this way, it's more simplified by handling both authentication and authorization with K-OAuth. Also, the credentials would never be transferred over the network.

7.1 Example case with K-OAuth

Figure 8 describes the same example from section *Example case with OpenID+OAuth* with work flow of authentication process with K-OAuth. In this case, the OpenID is replaced with Kerberos and hence K-OAuth. It could be seen that, interactions are similar until the authentication process in comparison with the OpenID+OAuth combination.



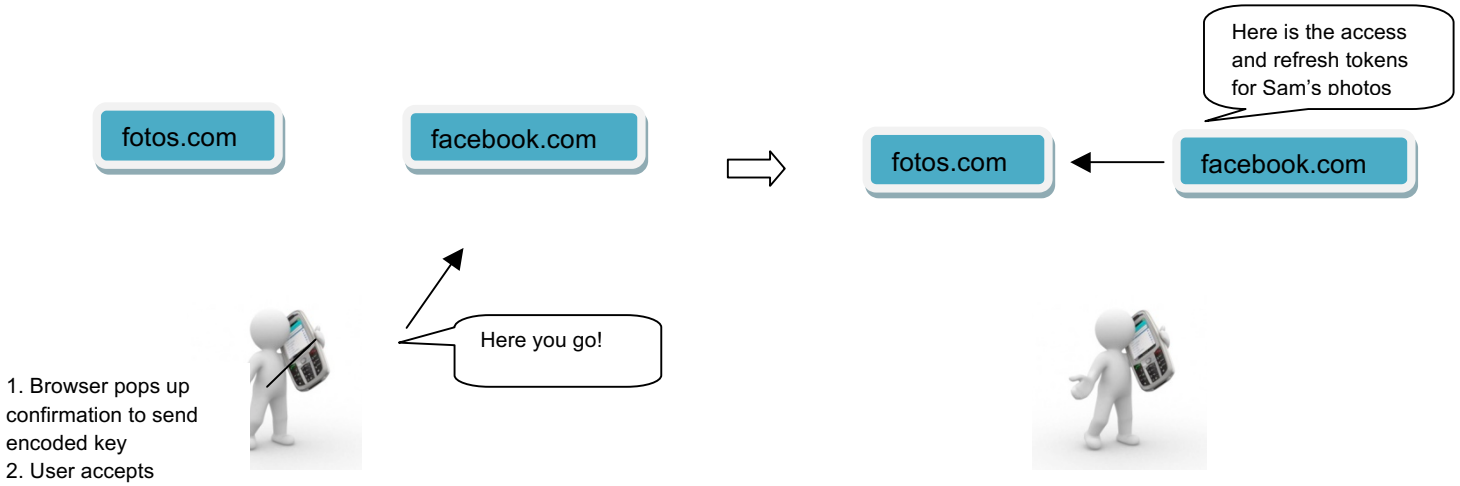


Figure 8 K-OAuth example case

When Sam is prompted to prove the identity, Sam is given options as to choose which way to prove it. Sam chooses K-OAuth then Sam is redirected again to a prompt where Sam gets a confirmation message to use the existing one if Sam hasn't created one yet. Sam has a possibility to create one right way by the K-OAuth slave residing in the device to provide the password. It's worth noting that, the password provided by Sam is neither sent to the browser nor over Internet. The password just resides in an uncommon place maintained by K-OAuth slave in the user device itself. The client residing in the device takes care of generating the encrypted temporary K-OAuth password which is sent over the Internet. The user has an option to perform this generation every time or store it locally.

7.2 K-OAuth in detail

7.2.1 K-OAuth Setup

The Figure 9 describes the K-OAuth in blocks. The blocks are indicated for clarity and better understanding. In real world they are not necessarily physical entities, they could co-exist.

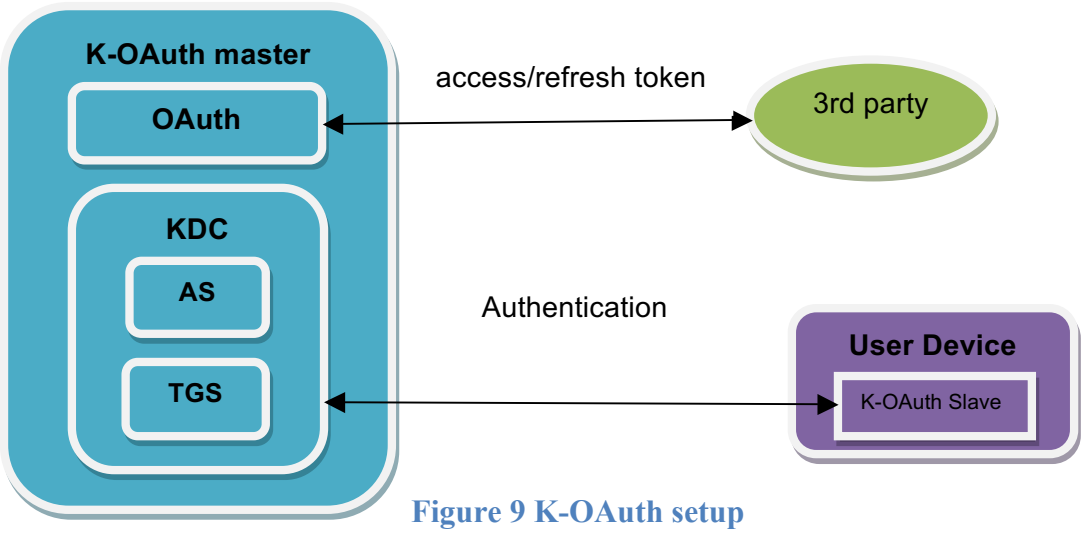


Figure 9 K-OAuth setup

The K-OAuth slave in the user device talks directly to KDC part of K-OAuth master and RP or some 3rd party servers which needs to access the protected user resource, communicates directly with the plain OAuth section of the K-OAuth master.

7.2.2 K-OAuth slave

The K-OAuth slave resides in the user's device. The slave acts an interface between the user's device and the Authentication server. It could be a dedicated process or service which runs in the background within the user's device. The slave is not restricted to web-based authentication but also performs negotiation with any other K-OAuth based authentication. However, the scope of the slave for web-based negotiations is only discussed here. The slave could interact as a standalone application or could be plug-in embedded in the device's browser.

8. K-OAuth explained

In this section, a detailed explanation of K-OAuth is discussed. A K-OAuth transaction is defined as the collection of interactions between the client and server with the same transaction identity. The K-OAuth transactions do not hold any state and hence would require a transaction identity provided by the K-OAuth master server after the first interaction with the client. The K-OAuth transactions involve critical information exchanged over HTTP post header.

8.1 K-OAuth transaction

A request for K-OAuth access token is initiated when the client needs to obtain an authorization from the resource owner. This authorization is expressed in the form of an authorization grant which the client uses to request the access token. K-OAuth defines two grant types which is modified version of authorized code grant and implicit grant inherited from OAuth2.0. The grant types defined in K-OAuth are pre-emptive approach (active) and lazy approach. There are also provisions for extension mechanism for defining additional grant types.

8.1.1 K-OAuth requests and responses

a. Authentication/Authorization request

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format. This pretty much resembles the format of OAuth2.0 authentication request.

response_type	REQUIRED. Value MUST be set to “init”
client_id	OPTIONAL. Every client is given a unique ID from the K-OAuth master upon user-account creation
redirect_uri	OPTIONAL. The URI to redirect upon successful authentication
state	RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The K-OAuth master server includes this value when redirecting the user-agent back to the client. The

parameter is used for preventing cross-site request forgery

b. Authentication/Authorization response

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format**.**

koauth_code	REQUIRED. The K-OAuth code generated by the authorization server. The authorization code expires shortly after it is issued to mitigate the risk of leaks. The maximum lifetime of authorization code is 10 minutes. The client must not use the authorization code more than once. If an authorization code is used more than once, the authorization server denies the request and revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.
state	REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.
id	REQUIRED. Defines the transaction identity for the client and server for the context awareness.

c. Two step token authentication request and responses

The two step token authentication is performed similar to the Kerberos authentication with the keys which are encrypted being passed over HTTP in JSON format. Based on the parameters beginning with “koauth_” the server understands the phase (step) of the authentication transaction.

The KOAuth server makes a request to the client by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body.

grant_type	REQUIRED. Value MUST be set to “active” or “lazy”

koauth_tgt_client	REQUIRED. Defines the TGT encrypted with client secret.
koauth_tgs	REQUIRED. Defines the TGT encrypted with client secret and TGS secret.
id	REQUIRED. Defines the transaction identity for the client and server for the context awareness.

The client responds to the K-OAuth server by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body.

grant_type	REQUIRED. Value MUST be set to “active” or “lazy”
koauth_tgt_tgs	REQUIRED. Defines the TGT encrypted with TGS secret.
koauth_id_tgt	REQUIRED. Defines the client identity encrypted with TGT.
id	REQUIRED. Defines the transaction identity for the client and server for the context awareness.
failed_redirect_uri	OPTIONAL. The URI to redirect upon failed authentication

Further on, as a second phase of the authentication process, the KOAuth server makes a request to the client by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body.

grant_type	REQUIRED. Value MUST be set to “active” or “lazy”
koauth_cstkt_res	REQUIRED. Defines the Client-server ticket encrypted with resource-server key.
koauth_cstkt_tgt	REQUIRED. Defines the Client-server ticket encrypted with TGT key.
id	REQUIRED. Defines the transaction identity for the client and server for the context awareness.

The client then responds to the K-OAuth server by adding the following parameters using the "application/x-www-form-urlencoded" format in the HTTP request entity-body.

grant_type	REQUIRED. Value MUST be set to “active” or “lazy”

koauth_cstkt_res	REQUIRED. Defines the Client-server ticket encrypted with resource-server key.
koauth_id_cstkt	REQUIRED. Defines the client identity encrypted with client-server ticket.
id	REQUIRED. Defines the transaction identity for the client and server for the context awareness.
failed_redirect_uri	OPTIONAL. The URI to redirect upon failed authentication

d. Authentication/Authorization failure (error) responses

The failure responses resemble the same as OAuth2.0. If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server informs the resource owner of the error, and does not automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI.

error	REQUIRED. A single error code from the following:	
	invalid_request	The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed
	unauthorized_client	The client is not authorized to request an authorization code using this method
	access_denied	The resource owner or authorization server denied the request
	unsupported_response_type	The authorization server does not support obtaining an authorization code using this method.

	invalid_scope	The requested scope is invalid, unknown, or malformed.
	server_error	The authorization server encountered an unexpected condition which prevented it from fulfilling the request.
	temporarily_unavailable	The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server.
error_description		OPTIONAL. A human-readable UTF-8 encoded text providing additional information, used to assist the client developer in understanding the error that occurred.
error_uri		OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
state		REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the client.

8.1.2 K-OAuth HTTP negotiations

A typical K-OAuth request/response could be similar to the ones presented in the OAuth2.0 specification. All attributes related to K-OAuth will prefix '*koauth*' to the key of the JSON requests/responses. There are some samples of successful request/response of K-OAuth negotiations. The failure or redirections are similar as in the OAuth2.0 specification.

Successful Request/Response	Description

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
"koauth_tgt_client":"2YotnFZFEjr1cMWpAA",
"koauth_tgs":"STY790ZFEjr1zCs23sdfdSH",
"token_type":"example",
"expires_in":3600,
"example_parameter":"example_value"
}
	TGT encrypted with client secret [koauth_tgt_client] and TGS secret [koauth_tgs] (OAuth_TGT)

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
"koauth_tgt_tgs":"FPOuin23SBNGHJUIhyomj",
"koauth_id_tgt":"23gkbGKeso23lLKJouf7",
"token_type":"example",
"expires_in":3600,
"example_parameter":"example_value"
}
	TGT encrypted with TGS secret [koauth_tgt_tgs] and ID encrypted with TGT [koauth_id_tgt] (OAuth_TGT+ID)

HTTP/1.1 200 OK Content-Type: application/json;charset=UTF-8 Cache-Control: no-store Pragma: no-cache { **"koauth_cstkt_res":"QWGas789DHBFiu2899",** **"koauth_cstkt_tgt":"lpgHKGI78dkNBIqwER",**	Client-server ticket encrypted with resource-server key [koauth_cstkt_res] and TGT [koauth_cstkt_tgt] (OAuth_client_to_server)

```
  "token_type":"example",
  "expires_in":3600,
  "example_parameter":"example_value"
}
```

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache
{
  "koauth_cstkt_res":"QWGas789DHBFiu2899",
  "koauth_id_cstkt":"yukRFF689giWBVCVa7",
  "token_type":"example",
  "example_parameter":"example_value"
}
```

Client-server ticket encrypted with resource-server key [koauth_cstkt_res] and ID encrypted with client-server ticket [koauth_id_cstkt] (OAuth_client_to_server+ID)

8.2 K-OAuth Operational flow

Here is an explanation on how the operations take place during authentication in a K-OAuth system. The user interacts with the user's device loaded with K-OAuth slave modules. The Authentication Server contains the K-OAuth master modules and user's protected resource.

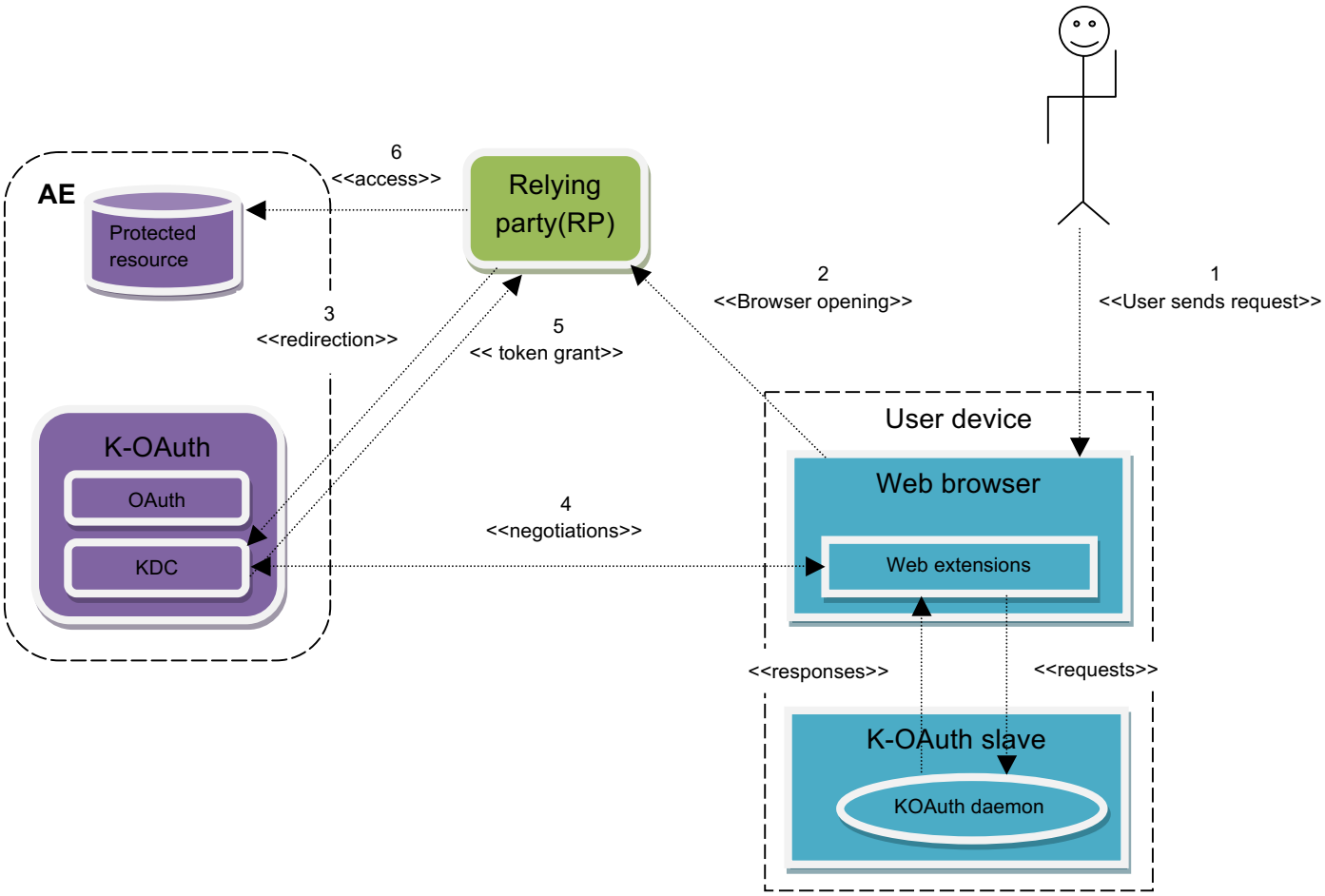


Figure 10 K-OAuth operational flow

The Figure 10 shows the operational flow of how K-OAuth negotiation takes place. The Relying party is the same entity which is defined in the OpenID2.0 and OAuth2.0 specifications. The authenticating and authorizing entity (AE) shown in the left side contains the K-OAuth master implementation and the user's protected resource. They do not necessarily reside in the same server. Also, at the right, there are only two blocks, web browser and K-OAuth slave

depicted in the user's device for convenience. The K-OAuth slave consists of K-OAuth daemon and some other packages and libraries related to Kerberos.

Step – 1 User requests the browser in the user's device which could have been an advertisement of the RP in a 3rd party website or by some means.

Step – 2 The browser opens the relying party web page where the user requests for a service which needs authorization from a reliable entity.

Step – 3 The RP redirects the user to reliable entity, the user has to authenticate if it wasn't done already or if the previous one was expired. The user chooses K-OAuth authentication/authorization mechanism.

Step – 4
- The K-OAuth server sends a TGT encrypted with client secret [koauth_tgt_client] (known to the user) and TGS secret [koauth_tgs] (not known to user) as a JSON response.
- The web extensions communicate with the K-OAuth daemon and return a TGT encrypted with TGS secret [koauth_tgt_tgs] with ID encrypted with TGT [koauth_id_tgt] as JSON response.
- Furthermore, the K-OAuth server sends a client-server ticket encrypted with resource-server key [koauth_cstkt_res] and TGT [koauth_cstkt_tgt].
- Upon receiving that, the K-OAuth slave responds with client-server ticket encrypted with resource-server key [koauth_cstkt_res] and ID encrypted with client-server ticket [koauth_id_cstkt].
- The user is then shown a confirmation dialog to authorize.

Step – 5 After the user confirms, the K-OAuth then sends the tokens to the RP. This access/refresh token granting happens as similar in OAuth2.0

Step – 6 The RP is now able to accesses the user's protected resources.

8.3 K-OAuth Approaches

A detailed explanation of different K-OAuth approaches is explained in this section.

8.3.1 Pre-emptive approach (Active)

In this kind of K-OAuth authorization, the user pre-emptively provides the encrypted key to the client which identifies the user and authenticity of the client.

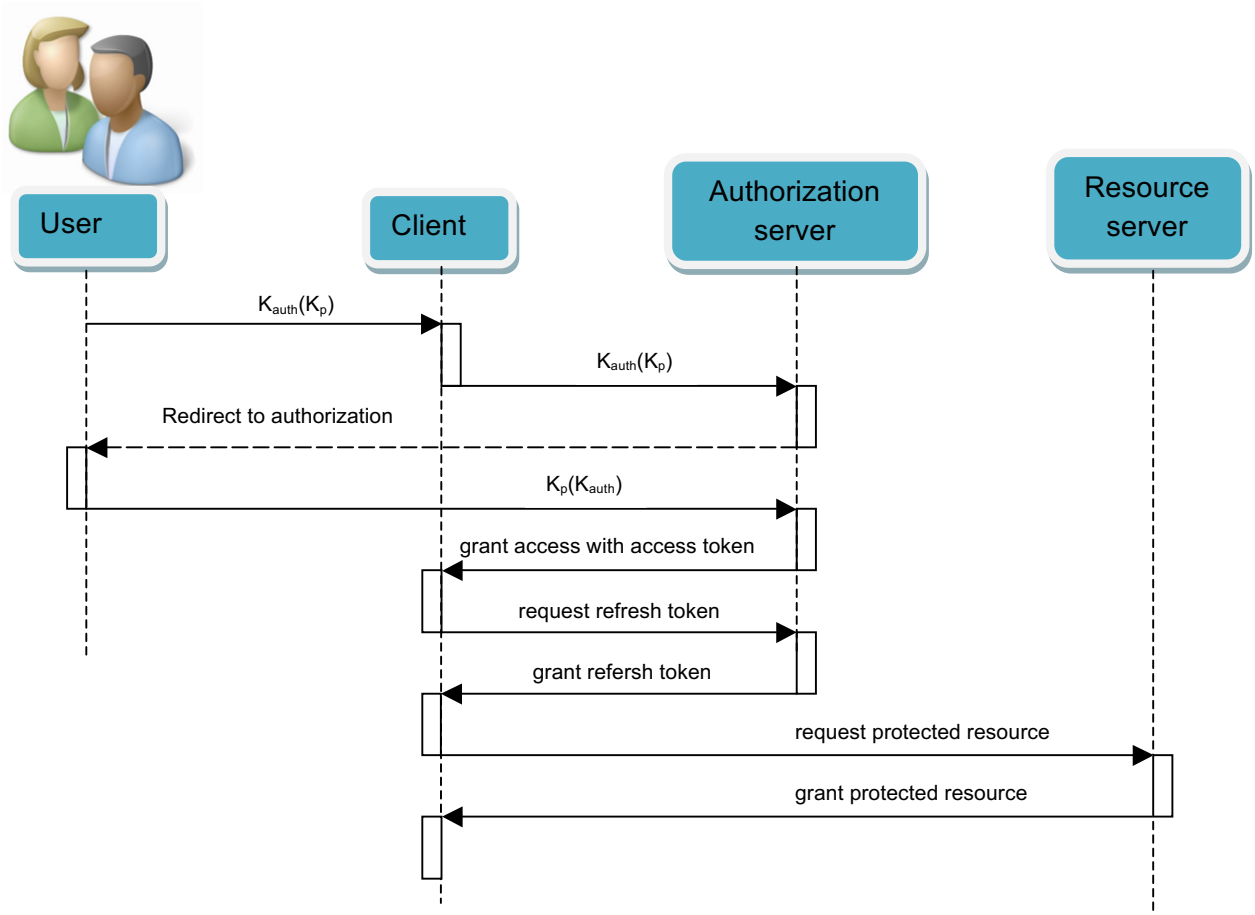


Figure 11 K-OAuth pre-emptive approach

1. The user provides a time stamped authentication key (K_{auth}) encrypted with user's password key (K_p) to the client (consumer) who needs access to protected resources present in the resource server (service provider).
2. The client redirects the user to Authorization Entity where the user provides password key (K_p) encrypted with same time bound temporary key (K_{auth}).
3. The service provider already has the password key (K_p) and hence could decrypt to obtain the temporary key (K_{auth}).
4. The service provider could also verify the authenticity of the user by decrypting the password key (K_p) and verifying if it matches.
5. Upon success, the service provider provides the consumer with an access token and refresh token for accessing the protected resources existing in the resource server.

8.3.2 Lazy approach

This is similar to the authentication/authorization implemented in OpenID+OAuth combined suite. In this case, the OpenID is being replaced with K-OAuth and authorization section being unaltered. This method starts with the plain request to the client from the user which is forwarded to authorization server as an authorization request. Compared to the previous approach, the K-OAuth server should check the authenticity of the user and the client in this case before the authorization process begins.

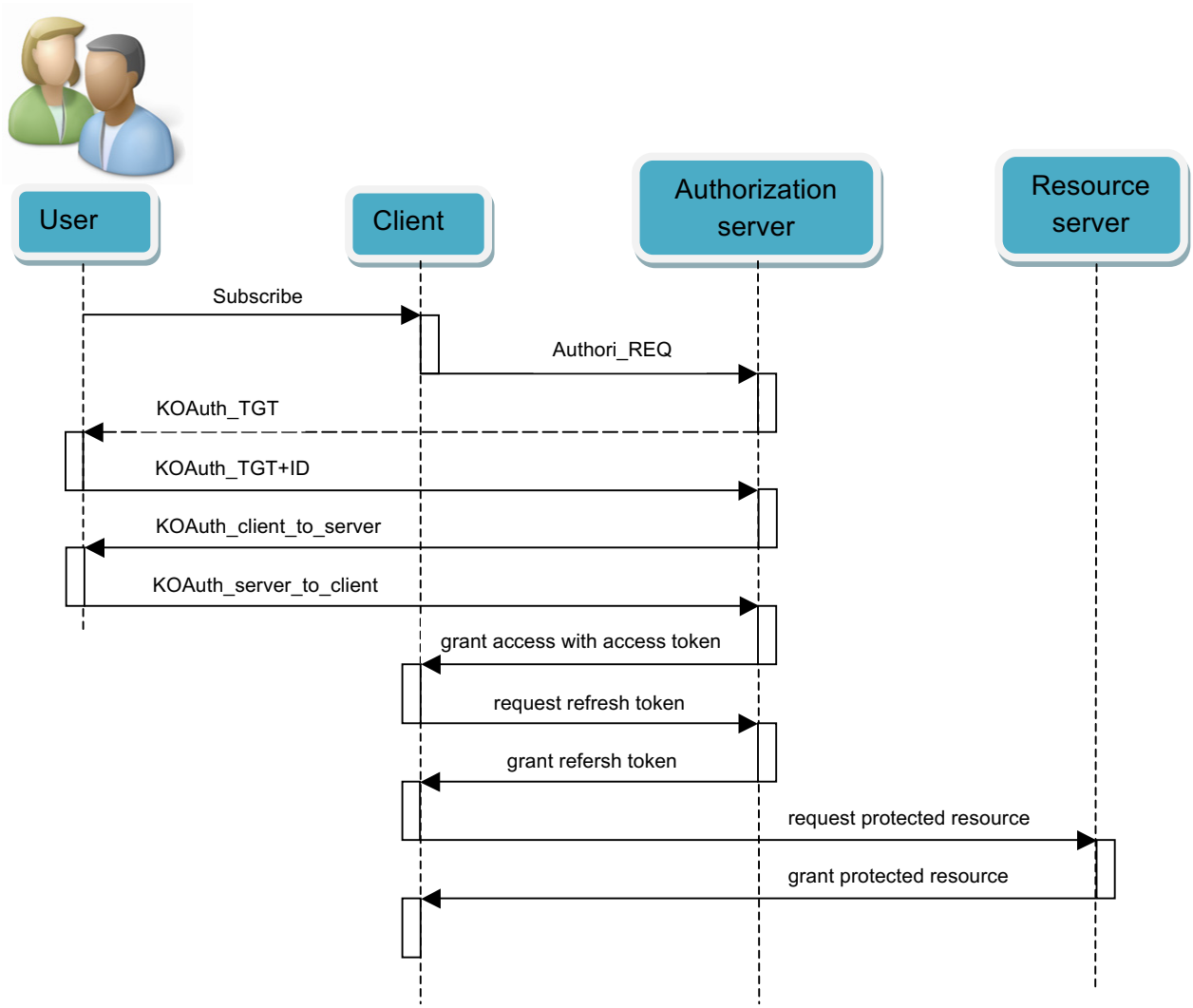


Figure 12 K-OAuth lazy authorization

Authori_REQ	Authorization request
KOAuth_TGT	TGT encrypted with client secret and TGS secret
KOAuth_TGT+ID	TGT encrypted with TGS secret with ID encrypted with TGT
KOAuth_client_to_server	client-server ticket encrypted with resource-server key and TGT
KOAuth_client_to_server+ID	client-server ticket encrypted with resource-server key and ID encrypted with client-server ticket

8.4 K-OAuth pros and cons

The K-OAuth is based on OAuth2.0 and inherits all the pros and cons of it. Also, there are some benefits and drawbacks with Kerberos based authentication. This section discusses this in detail.

8.4.1 K-OAuth strengths

The biggest advantage of moving towards K-OAuth would be that, the K-OAuth bundles the authentication and authorization protocols into a single entity for the ease of administration. Also the end user does not have to be aware of the phases and complexities with Kerberos authentication and is embedded native with the application.

K-OAuth also addresses problems relating to credentials sent across various domains in internet. As discussed in previous sections of this document, the credentials traversing across domains are the primary cause for the problem. In K-OAuth the credentials are never sent across the network, encrypted or in plain text format. Secret keys are only passed across the network in encrypted form. Hence, a miscreant snooping and logging conversations on a possibly insecure network cannot deduce from the contents of network conversations enough information to impersonate an authenticated user or an authenticated target service [36].

Client and server systems mutually authenticate at each step of the process, both the client and the server systems may be certain that they are communicating with their authentic counterparts espeically with respect to the transaction id exchanged at the beginning of the transaction. The tickets passed between clients and servers in the Kerberos authentication model include timestamp and lifetime information. This allows K-OAuth clients and servers to limit the duration of their users' authentication. While the specific length of time for which a user's authentication remains valid after his initial ticket issued is implementation dependent,

K-OAuth systems typically use small enough ticket lifetimes to prevent brute-force and replay attacks. In general, no authentication ticket should have a lifetime longer than the expected time required to crack the encryption of the ticket [36].

The K-OAuth client agent residing in the user’s device has some intelligence which gives some ease to the user and also to the implementation. Benefits like safe caching of credentials, caching

of server behaviours would improve the user experience and the K-OAuth server pages could contain a generic implementation in specific to device and operating systems.

8.4.2 K-OAuth weeknesses

The K-OAuth uses a mutual authentication model, it is necessary for both client machines and service providers (servers) to be designed with K-OAuth authentication in mind. Though K-OAuth is derived from OAuth2.0, the already implemented OAuth2.0 servers are not said to be functional with K-OAuth. This is equivalent to adopting a new protocol from the perspective of implementation.

K-OAuth handles the problems relating to credentials passed over internet but this system is still vulnerable for attacks as there are circumstances when the user still has to type the credentials. Malwares like keystroke logger could still hijack the user's credentials in some cases as there is credentials transferred to the server the first time. The K-OAuth slave residing in the user's device is also potential for internet attacks.

The K-OAuth slave is also a application and hence needs updates based on the improvements in the server side. This would cause a more maintenance for administrators as this needs to be pushed to all users registered with server. There can be also problems related to backward compatibility.

9. Conclusion

A description and analysis of the OpenID Single Sign-On protocol, OAuth2.0 authorization protocol, Kerberos V5 protocol and eventually Kerberos-OAuth protocol for authentication and authorization are discussed. The model of OpenID seems to be a suitable Single Sign-On solution for the Internet of today. It has remarkable usability properties and the concept of extensions makes it very flexible. There are a lot of drawbacks with OpenID and internet phishing being one of the serious caused due to decentralized authentication combined with transporting credentials over internet.

Extending OAuth2.0 to perform authentication will allow simplicity in protocol and setting up the system. This combined suite could seamlessly integrate authentication and authorization so that it's much more flexible and scalable. It inherits all the benefits and simplicity from OpenID+OAuth combination. One could understand that, they are just extensions for OAuth2.0. The K-OAuth authentication addresses some security issues with OpenID by preventing the passing of user credentials over Internet, thereby avoiding attacks like Internet phishing. An attempt to re-implement the strategies of a successful matured authentication protocol like Kerberos with newer protocols like OAuth2.0 would overcome problems related to past and could be an attempt to use solutions for already solved problems.

In future work, the plan is to abstract the KDC much more and make it native to K-OAuth. In which case, there is a possibility of getting rid of the slave which is currently a thin layer in the user's device. The server could provide a native plugin which could be dynamically downloaded and executed on fly instead of a dedicated background process serving the K-OAuth master. With the advent of HTML5, there are more possibilities for the browser to safely interact with user's device applications or libraries. In this way, there will be no need for regular software updates pushed to the user nodes. The problem with typing credentials occurs in a K-OAuth only during the time of setup or first signup. This is a way for the K-OAuth server to know the user's genuine credentials. Though this is going to be very rare, there can be other means to transfer user's credentials in another manner apart from the hand typing it through browser. Also there are plans to expand this K-OAuth authentication for mechanisms other than web-based. The same mechanism could be implemented for authenticating and authorizing command line

interface. Kerberos kind of authentication is already well established for command-line authentications.

10. References

[1] OpenID specification: http://OpenID.net/specs/OpenID-authentication-2_0.html

[2]. Mashima D, Ahamad M. In: Towards a user-centric identity-usage monitoring system. Internet monitoring and protection, 2008. ICIMP '08. the third international conference on; ; 2008. p. 47-52.

[3] OAuth2.0 (draft) RFC: http://tools.ietf.org/html/draft-ietf-oauth-v2-22

[4] Oauth1.0 RFC: http://art.tools.ietf.org/html/rfc5849

[5]. Wang Bin, Huang He Yuan, Liu Xiao Xi, Xu Jing Min. In: Open identity management framework for SaaS ecosystem. e-business engineering, 2009. ICEBE '09. IEEE international conference on; ; 2009. p. 512-7.

[6]. Prehofer C, van Gurp J, Stirbu V, Satish S, Tarkoma S, di Flora C, et al. Practical web-based smart spaces. Pervasive Computing, IEEE. 2010;9(3):72-80.

[7]. Recordon D, Reed D. In: OpenID 2.0: A platform for user-centric identity management. Proceedings of the second ACM workshop on digital identity management; Alexandria, Virginia, USA. New York, NY, USA: ACM; 2006. p. 11-6.

[8]. Huang C, Ma S, Chen K. Using one-time passwords to prevent password phishing attacks. Journal of Network and Computer Applications. 2011 7;34(4):1292-301.

[9]. HwanJin Lee, InKyung Jeun, Kilsoo Chun, Junghwan Song. In: A new anti-phishing method in OpenID. Emerging security information, systems and technologies, 2008. SECURWARE '08. second international conference on; ; 2008. p. 243-7.

[10] Setting up OpenID: http://code.google.com/apis/accounts/docs/OpenID.html

[11]. Lynch L. Inside the identity management game. Internet Computing, IEEE. 2011;15(5):78-82.

[12]. Lunt SJ. In: Using kerberos in operations systems. Global telecommunications conference, 1991. GLOBECOM '91. 'countdown to the new millennium. featuring a mini-theme on: Personal communications services; ; 1991. p. 687,693 vol.1.

[13] Kerberos RFC: http://www.ietf.org/rfc/rfc4120.txt

[14] Shieh S, Yang W. An authentication and key distribution system for open network systems. SIGOPS Oper.Syst.Rev. 1996 April;30(2):32-41.

[15] Jae-Hwe You, Moon-Seog Jun. In: A mechanism to prevent RP phishing in OpenID system. Computer and information science (ICIS), 2010 IEEE/ACIS 9th international conference on; ; 2010. p. 876-80.

[16] Qingxiang Feng, Kuo-Kun Tseng, Jeng-Shyang Pan, Peng Cheng, Chen C. In: New anti-phishing method with two types of passwords in OpenID system. Genetic and evolutionary computing (ICGEC), 2011 fifth international conference on; ; 2011. p. 69-72.

[17] Hsu, Francis. et al, "WebCallerID: Leveraging cellular networks for Web authentication" Journal of Computer Security vol. 19, no. 5 (2011), p. 869, 2011

[18] Kerberos: An Authentication Service for Computer Networks: http://gost.isi.edu/publications/Kerberos-neuman-tso.html

[19] Shen, Yang et al. "Research and design of single sign-on based on Kerberos protocol", Computer Engineering and Design vol. 32, no. 7 (Jul 2011), p. 2249

[20] "Google Embraces OAuth Authentication For Apps", Informationweek - Online (Sep 28, 2010), p. n/a, 2010

[21]. Ding Chu, Qing Liao, Jingling Zhao. In: Open identity management framework for mashup. Web society (SWS), 2010 IEEE 2nd symposium on; ; 2010. p. 378-82.

[22]. Yating Hsu, Lee D. In: Authentication and authorization protocol security property analysis with trace inclusion transformation and online minimization. Network protocols (ICNP), 2010 18th IEEE international conference on; ; 2010. p. 164-73.

[23]. Wenjun Wu, Hui Zhang, ZhenAn Li. In: Open social based collaborative science gateways. Cluster, cloud and grid computing (CCGrid), 2011 11th IEEE/ACM international symposium on; ; 2011. p. 554-9.

[24] Inoue T, Asakura H, Sato H, Takahashi N. In: Key roles of session state: Not against REST architectural style. Computer software and applications conference (COMPSAC), 2010 IEEE 34th annual; ; 2010. p. 171-8.

[25] Phishing in OpenID: http://www.identityblog.com/?p=659

[26] Boldyreva A, Kumar V. Provable-security analysis of authenticated encryption in kerberos. Information Security, IET. 2011;5(4):207-19.

[27]. Backes M, Cervesato I, Jaggard A, Scedrov A, Tsay J. Cryptographically sound security proofs for basic and public-key kerberos. 2006;4189:362-83.

[28]. Hellewell PL, van der Horst TW, Seamons KE. In: Extensible pre-authentication kerberos. Computer security applications conference, 2007. ACSAC 2007. twenty-third annual; ; 2007. p. 201-10.

[29]. Chunxiao Fan, Shuai Yang, Junwei Zou, Xiaoying Zhang. A New Secure OpenID Authentication Mechanism Using One-Time Password (OTP). Wireless Communications, Networking and Mobile Computing (WiCOM), 7th International Conference on 23-25 Sept. 2011; p. 1-4.

[30]. Watanabe R, Tanaka T. Federated Authentication Mechanism using Cellular Phone - Collaboration with OpenID. Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on 27-29 April 2009; p. 435-442.

[31]. Al-Janabi S.T.F., Rasheed M.A.-S. Public-Key Cryptography Enabled Kerberos Authentication. Developments in E-systems Engineering (DeSE), Dec. 2011; p. 209-214.

[32]. Kerberos User's guide: http://web.mit.edu/kerberos/krb5-latest/krb5-1.10/doc/krb5-user.html

[33]. OpenID OAuth hybrid extension (draft): http://svn.openid.net/repos/specifications/oauth_hybrid/1.0/trunk/openid_oauth_extension.html

[34]. Xiangwu Ding, Junyin Wei. In: A Scheme for Confidentiality Protection of OpenID Authentication Mechanism. Computational Intelligence and Security (CIS), 2010 International Conference on; ;2010. p. 310-314.

[35]. Junyin Wei, Mingxi Zhang, Xiangwu Ding, Ying Wang. In: Research on Multi-Level Security Framework for OpenID. Electronic Commerce and Security (ISECS), 2010 Third International Symposium on; ;2010. p. 393-397.

[36]. Kerberos strengths and weeknesses: "http://www.duke.edu/~rob/kerberos/kerbasnds.html"