University of Piraeus

Department of Digital Systems

Postgraduate Program «Digital Systems Security»

Academic Year 2017-2018

(ΨΣ-ΑΦ-888) – MSc Dissertation

Combination of the PEAP Protocol with EAP-OpenID Connect

Bolgouras Vaios
MTE1624, vaiompolgoyras@ssl-unipi.gr

Supervisor Professor

Dr. Xenakis Christos

Piraeus, March 2018

Special Thanks to Dr. Xenakis and Dr. Dadoyan for their guidance and to my fellow student Panagioti Bountaka for his help.

Contents

Table of Figures

Abstract

Connecting to a wireless network in most cases requires either a password or a certificate. Enabling authentication using a social media account within a Wi-Fi network implies that a captive portal or another enterprise solution should be implemented. In this Thesis we propose an alternative approach that combines Extensible Authentication Protocol (EAP) and OpenID Connect (OIDC) protocol in order to create a new EAP method for authentication in Wi-Fi networks with social media or email accounts. More specifically we propose the encapsulation of OIDC protocol messages within EAP packets through a secure TLS tunnel, thus the concept is named PEAP with EAP-OIDC.

Introduction

Nowadays, in the modern world, it is required from us to have accounts everywhere. The majority of the accounts support a username-password combination. According to the online survey, in which more than 2,000 English-speaking adults participated, the average person has 27 discrete online logins [14]. 27 username-password combinations are way too much for one person to remember. Hence, this leads users to use the same password over and over again. Technologies like OAuth [9] and OIDC [8] were created in order to mitigate this risk by letting users take advantage of their social media or email accounts instead of creating new accounts in every service. This resulted in a new role for the social media and email providers, to act as identity providers (IdPs). Most of the restaurants, coffee shops, hotels, universities, etc. also offer free Wi-Fi on top of their other services. Some of them use an open Wi-Fi, while others use a password protected Wi-Fi which means, one single password for every user. On the one hand, if the Wi-Fi is not password protected then it is easy to connect, but it has several security issues, which are beyond the scope of this thesis. On the other hand, if the Wi-Fi is protected by a password, the user should look for signs with the password written on them or ask someone about it. We all know that this is a time consuming procedure, that nobody looks forward to. Imagine the convenience of having the ability to, instead of looking for the password, use a social media account or email credentials and get access on the Wi-Fi network. The application of such a scheme could also be implemented in corporate environments, to make it easy - only - for the employees to login their network.

There are already some solutions that offer authentication in Wi-Fi networks with social or email accounts, like captive portal, but they are not secure enough, since the user obtains - restricted - access on the network before the authentication takes place. In this thesis we propose a theoretical method which combines EAP [1] and OIDC in order to achieve authentication in Wi-Fi networks with social media or email accounts. This method will offer a high level of security, as the user obtains access on the network after the authentication. Moreover, in order to achieve a secure communication, we first establish a secure tunnel and then the EAP-OIDC procedure commences. The secure tunnel is deployed via the Protected EAP method.

EAP - OpenID Connect Foundations

OAuth 2.0

OAuth 2.0 is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as social media, email and HTTP service providers in general. It delegates the user authentication to the service that hosts the user account and authorizes a 3rd party application to access the user account. It defines four roles: Resource Owner i.e. the user, Client i.e. the application, Resource Server and Authorization Server. An abstract protocol flow is shown in the above figure.

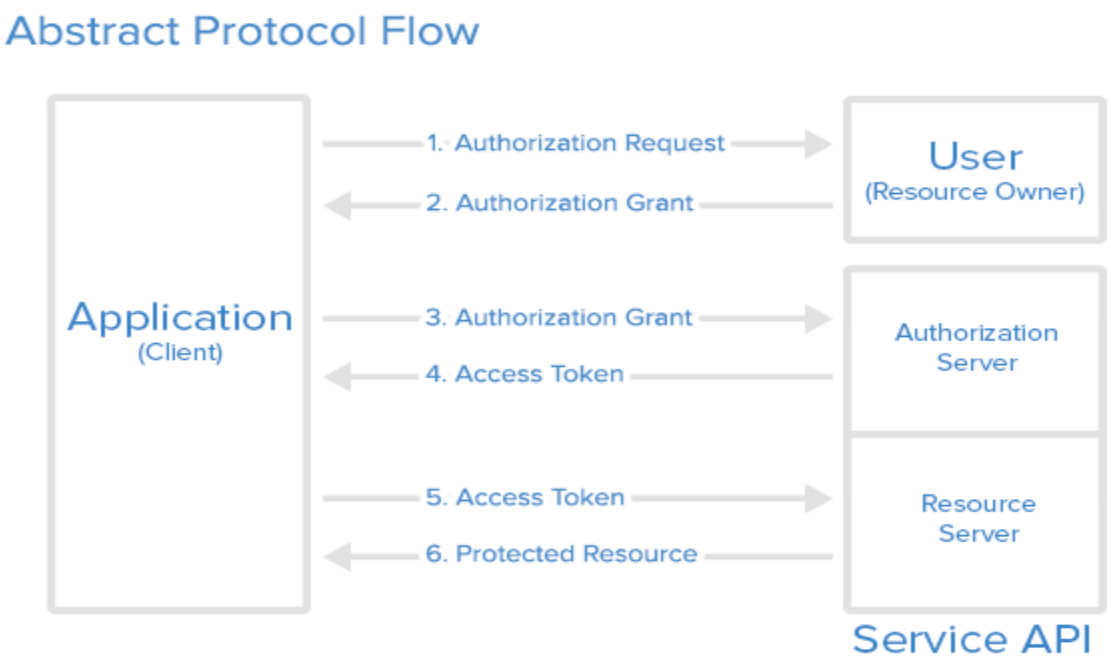


Figure 1: OAUTH2.0 overview

1. The *application* requests authorization to access service resources from the *user*
2. If the *user* authorized the request, the *application* receives an authorization grant
3. The *application* requests an access token from the *authorization server* (API) by presenting authentication of its own identity, and the authorization grant
4. If the application identity is authenticated and the authorization grant is valid, the *authorization server* (API) issues an access token to the application. Authorization is complete.
5. The *application* requests the resource from the *resource server* (API) and presents the access token for authentication
6. If the access token is valid, the *resource server* (API) serves the resource to the *application*

The real flow of this process will differ depending on the authorization grant type in use.

The use of OAuth within an application implies that it should be registered with the service. This is done through a registration form in the service's website. In this form the following information should be provided: application name, application website, redirect URI or callback URI.

The redirect URI is where the service will redirect the user after they authorize (or deny) the application, and therefore the part of the application that will handle authorization codes or access tokens.

Once the application is registered, the service will issue "client credentials" in the form of a client identifier and a client secret. The Client ID is a publicly exposed string that is used by the service API to identify the application, and is also used to build authorization URLs that are

presented to users. The Client Secret is used to authenticate the identity of the application to the service API when the application requests to access a user's account, and must be kept private between the application and the API.

Authorization Grant

The grant type flows determine how the Access Token are returned to the Client. The different grant types are:

- Authorization Code: Usually used with server-side application
- Implicit: Used with mobile apps or applications that runs on the user ‘s device.
- Resource Owner Password Credentials: Used with trusted applications, such as those owned by the service itself.
- Client Credentials: Used with applications API access.

Authorization Code Flow

The most commonly used grant type is the authorization code grant, because it is optimized for *server-side applications*, where source code is not publicly exposed, and *Client Secret* confidentiality can be maintained. This is a redirection-based flow, which means that the application must be capable of interacting with the *user-agent* (i.e. the user's web browser) and receiving API authorization codes that are routed through the user-agent. The authorization code flow is described below:

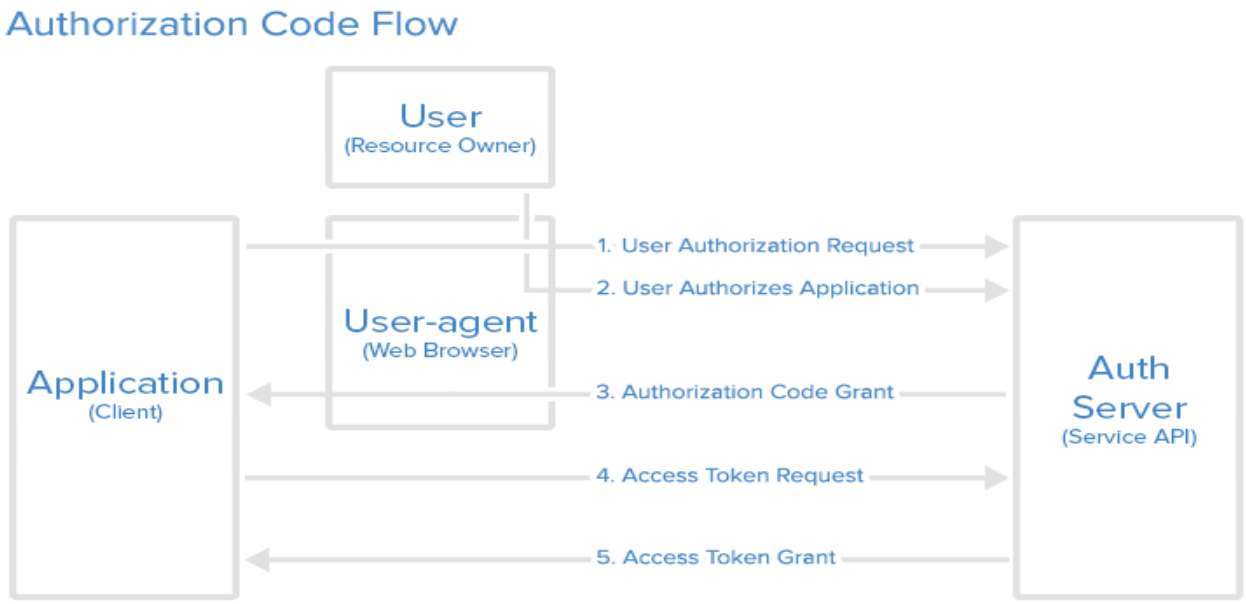


Figure 2: Authorization Code Flow

1) The user is given an authorization code link which have the following components:

- https://www.example.com/ v1/oauth/authorize: the API authorization endpoint
- client_id=id: the application's client ID
- redirect_uri=callback_URI: where the service redirects the user-agent after an authorization code is granted
- response_type=code: specifies that the authorization code grant type is used
- scope=read: specifies the level of access that the application is requesting
- Full example: https://www.oauth.example.com/v1/oauth/authorize?response_type=code&client_id=id&redirect_uri=callback_URL&scope=read

2) The user clicks the link, firstly they must log in to the service, in order to authenticate their identity (unless they are already logged in). Then they will be prompted by the service to *authorize* or *deny* the application to access their account.
3) If the user chooses the "authorize" option, the service redirects the user-agent to the application redirect URI, which was specified during the client registration, along with an *authorization code*. Example: https://application.com/callback?code=AUTHORIZATION_CODE
4) The application requests an access token from the API, by passing the authorization code along with authentication details, including the client secret, to the API token endpoint.
5) For a valid authorization the API will send a response containing the access token to the application. It may use the token to access the user's account via the service API until the token expires or is revoked.

Implicit

The implicit grant type usually used for mobile apps and web applications, where the client secret confidentiality is not guaranteed. This grant type is also a redirection-based flow but the access token is given to the user-agent to forward to the application, hence it may be exposed to the user and other applications on the user's device. Also, this flow does not authenticate the identity of the application, and relies on the redirect URI to serve this purpose. The full process using implicit grant type is discussed below:

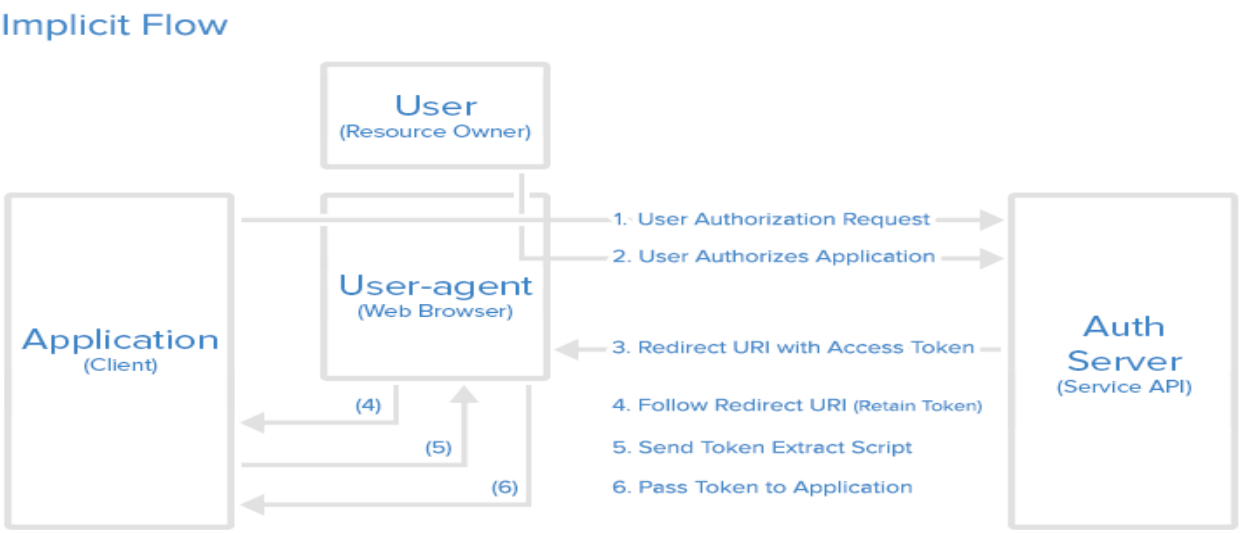


Figure 3: Implicit Flow

1) With the implicit grant type, the user is presented with an authorization link, that requests a token from the API. This link looks just like the authorization code link, except it is requesting a token instead of a code.
2) This step is the same with the authorization code. The user clicks the link, firstly they must log in to the service, in order to authenticate their identity (unless they are already logged in). Then they will be prompted by the service to ***authorize*** or ***deny*** the application access to their account.
3) If the user chooses the "authorize" option, the service redirects the user-agent to the application redirect URI, and includes a URI fragment containing the access token. It would look like this: https://application.com/callback#token=ACCESS_TOKEN
4) The user-agent follows the redirect URI but retains the access token.
5) The application returns a webpage that contains a script that can extract the access token from the full redirect URI that the user-agent has retained.
6) The provided script is executed by the user-agent and passes the extracted access token to the application. It may use the token to access the user's account via the service API until the token expires or is revoked.

Resource Owner Password Credentials

The user provides his service credentials (username and password) directly to the application, which uses the credentials to obtain an access token from the service. This grant type should only be enabled on the authorization server if other flows are not viable. Also, it should only be used if the application is trusted by the user.

After the user shares his credentials with the application, the application will then request an access token from the authorization server. This request might look like: https://www.oauth.example.com/token?grant_type=password&username=USERNAME&password=PASSWORD&client_id=CLIENT_ID

If the user credentials check out, the authorization server returns an access token to the application and the application is authorized.

Client Credentials

The client credentials grant type provides a way to the application of accessing its own service account. This might be useful if the application wants to update its registered description or redirect URI, or access other data stored in its service account via the API.

The application requests an access token by sending its credentials, its client ID and client secret, to the authorization server. This request might look like:
https://www.oauth.example.com/token?grant_type=client_credentials&client_id=CLIENT_ID&client_secret=CLIENT_SECRET

If the application credentials check out, the authorization server returns an access token to the application and the application is authorized.

OpenID Connect

OIDC is an OpenID Foundation standard. The OpenID Foundation formed in 2007 and is a non-profit international standardization organization of individuals and companies committed to enabling, promoting and protecting OpenID technologies. OIDC is published in 2014 and it is an identity layer on top of the OAuth 2.0 protocol. It enables clients to verify the identity of the user based on the authentication performed by an authorization server. In OIDC the roles differ a little from OAuth. The Resource Owner is called End-User, the Client is called Relying Party (RP), the Resource and Authorization server are called OpenID Provider (OP). The authentication in OIDC protocol can follow one of three paths. These paths are basically three flows which determine how the ID and Access token are returned to the client:

- Authorization code flow (response_type=code): This is the most commonly used flow, intended for traditional web apps as well as native/mobile apps. Involves an initial browser redirection to/from the OP for a user authentication and consent and then a second back channel request is used to retrieve the ID token.
- Implicit flow (response_type=id_token token): Usually used by the browser based apps that do not have a backend and the ID token is received directly.
- Hybrid flow: It is a combination of the two previous flows. This is the most rarely used which allows the app front-end and back-end to receive tokens separately from one another.

As it mentions before, the OIDC reuses the OAuth 2.0 protocol and parameters and extends it to introduce an Identity Layer through the following additions:

- Along with the access token, an ID token is returned, which is basically a JSON Web Token with identity claims (user information). The ID Token format is described in the next chapter.

- A UserInfo endpoint introduced, which returns basic profile attributes against the access token.

Authorization Code Flow

With the Authorization Code Flow all tokens are returned from the Token Endpoint. The Authorization Code Flow returns an Authorization Code to the RP, which can then exchange it for an ID Token and an Access Token directly.

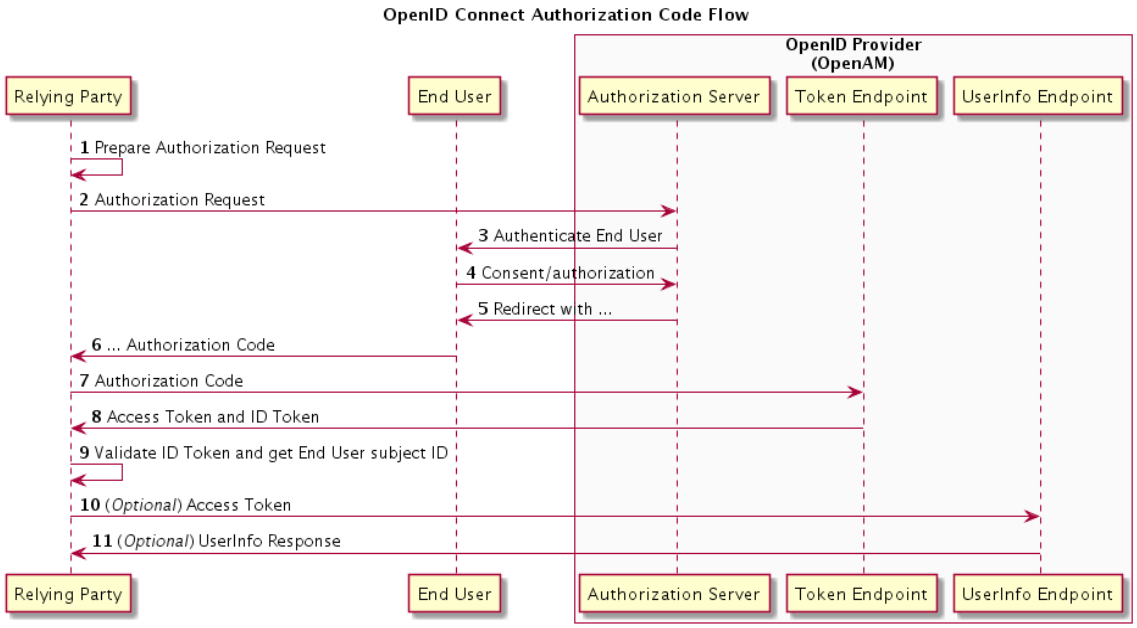


Figure 4: Authorization Code Flow

The above figure illustrates the Authorization Code Flow and the whole process is described below:

1. RP prepares an Authentication Request containing the desired request parameters.
2. RP sends the request to the Authorization Server.

   RP may use the HTTP GET or POST methods to send the Authorization Request to the Authorization Server. If the HTTP GET method is used, the request parameters are serialized using URI Query String Serialization (adding the parameters and values to the query component of a URL using the application/x-www-form-urlencoded format), if the HTTP POST method is used, the request parameters are serialized using Form Serialization (adding the parameter names and values to the entity body of the HTTP request using the application/x-www-form-urlencoded format).
3. Authorization Server Authenticates the End-User.

   If the request is valid, the Authorization Server attempts to Authenticate the End-User or determines whether the End-User is Authenticated, depending upon the request parameter values used. The Authorization Server may use different methods to Authenticate the End-User, e.g. username and password, session cookies, etc.
4. Authorization Server obtains End-User Consent/Authorization.

Once the End-User is authenticated, the Authorization Server MUST obtain an authorization decision before releasing information to the RP.

5,6. Authorization Server sends the End-User back to the RP with an Authorization Code.

7. RP requests a response using the Authorization Code at the Token Endpoint.

A RP makes a Token Request by presenting its Authorization Grant (in the form of an Authorization Code) to the Token Endpoint using the grant_type value authorization_code. The RP sends the parameters to the Token Endpoint using the HTTP POST method and the Form Serialization.

8. RP receives a response that contains an ID Token and Access Token in the response body.

After receiving and validating a valid and authorized Token Request from the RP, the Authorization Server returns a successful response that includes an ID Token and an Access Token. All Token Responses that contain tokens, secrets, or other sensitive information MUST include the following HTTP response header fields and values:

Header Name	**Header Value**
Cache-Control	no-store
Pragma	no-cache

The following is a successful example of a Token Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
 "access_token": "SlAV32hkKG",
 "token_type": "Bearer",
 "refresh_token": "8xLOxBtZp8",
 "expires_in": 3600,
 "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc
   yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5
   NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZ
   fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
   AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6q
   Jp6IcmD3HP99Obi1PRs-cwh3LO-p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
   NqeGpe-gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd
   QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoS
   K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4
   XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
}
```

9. RP validates the ID token and retrieves the End-User's Subject Identifier.

To validate the ID Token the RP MUST:

- Follow the validation rules in RFC 6749 (OAuth 2.0)
- Follow the ID Token validation rules
- Follow the Access Token validation rules

10 (Optional). RP requests a response using the Access Token at the Userinfo Endpoint.

11 (Optional). RP receives a response that contains user information in the response body.

The exchange of the above messages is done with HTTPS or HTTP protocol, with GET and POST methods.

Implicit Flow

With the Implicit Flow, all tokens are returned from the Authorization Endpoint. The token endpoint is not used. The Implicit Flow is mainly used by RPs implemented in a browser using a scripting language. The Access Token and ID Token are returned directly to the RP, which may expose them to the End User and applications that have access to the End User ‘s User Agent. The below figure shows the Implicit flow procedure.

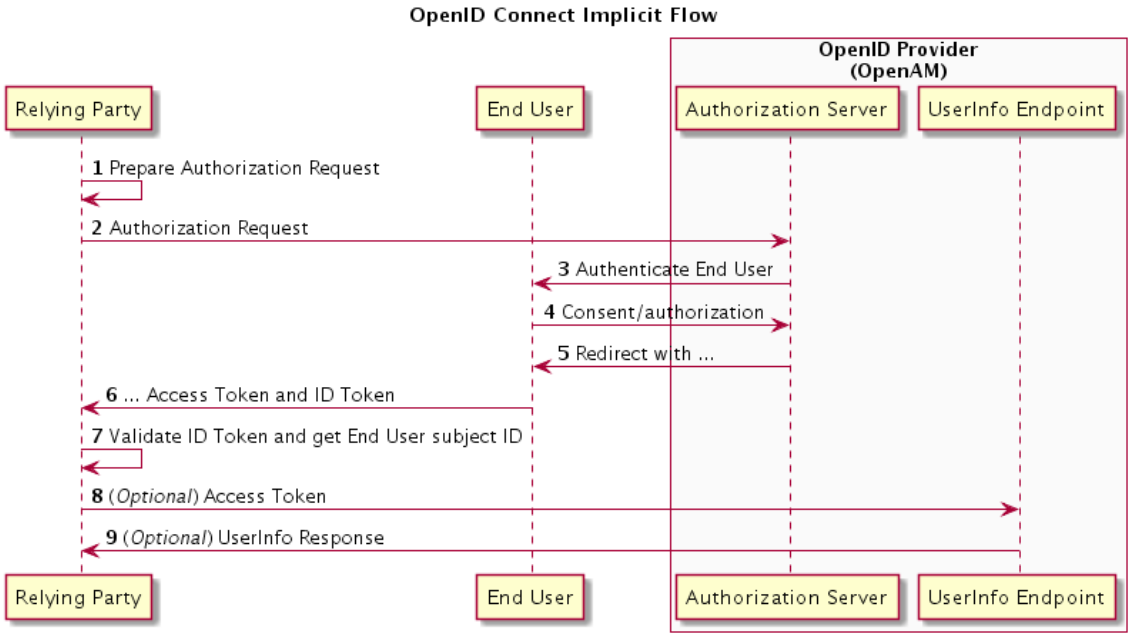


Figure 5: Implicit Flow

1. Client prepares an Authorization Request containing the desired request parameters.
2. RP sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5,6. Authorization Server sends the End-User back to the RP with an ID Token and, if requested, an Access Token.
7. RP validates the ID token and retrieves the End-User's Subject Identifier.
8 (Optional). RP requests a response using the Access Token at the Userinfo Endpoint.
9 (Optional). RP receives a response that contains user information in the response body.

The above requests and responses are made in the same manner as with the Authorization Code Flow. The exchange of the above messages is done with HTTPS or HTTP protocol, with GET and POST methods.

Hybrid Flow

When using the Hybrid flow, some tokens are returned from the Authorization endpoint and others are returned from the token endpoint. This method is not very common hence we will only present the whole procedure's steps.

The Hybrid Flow follows the following steps:

1. Client prepares an Authentication Request containing the desired request parameters.
2. Client sends the request to the Authorization Server.
3. Authorization Server Authenticates the End-User.
4. Authorization Server obtains End-User Consent/Authorization.
5. Authorization Server sends the End-User back to the Client with an Authorization Code and, depending on the Response Type, one or more additional parameters.
6. Client requests a response using the Authorization Code at the Token Endpoint.
7. Client receives a response that contains an ID Token and Access Token in the response body.
8. Client validates the ID Token and retrieves the End-User's Subject Identifier.

ID Token

Client applications receive the user 's identity with an ID Token which is the primary extension that OIDC makes to OAuth 2.0 to enable End-Users authentication. The ID Token is basically a JSON Web Token (JWT). JWT is an open, industry standard (RFC 7519) method that defines a compact and self-contained way for securely transmitting information between two parties as a JSON Object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**. JWT are useful for authentication, as when the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains. Also it is useful for information exchange, since it can be signed and you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

Anatomy of a JSON Web Token (JWT)

A JWT consists of three base 64 encoded strings separated by "." and it looks like aaaaaaa.bbbbbbbbbbb.cccccccc .

This three parts are:

- Header
- Payload
- Signature

Header

The header carries two parts, declaring the type (which is JWT) and the hashing algorithm to use (for example HMAC SHA256). So the header before encoded to base 64 look like:

{

“typ”: “JWT”,

“alg”: “HS256”

}

And in base 64 format: **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**

Payload

The payload will carry the JWT claims. This is where the information that we want to transmit and other information about the token. There are multiple claims that we can provide. This includes registered claim names, public claim names and private claim names.

Registered claims

Claims that are not mandatory whose names are reserved for us. These include:

- iss: The issuer of the token
- sub: The subject of the token
- aud: The audience of the token
- exp: This will probably be the registered claim most often used. This will define the expiration in NumericDate value. The expiration MUST be after the current date/time.
- nbf: Defines the time before which the JWT MUST NOT be accepted for processing
- iat: The time the JWT was issued. Can be used to determine the age of the JWT
- jti: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

Public claims:

These are the claims that we create ourselves like user name and other important information.

Private claims:

A producer and a consumer may agree to use claim names that are private.

The payload before the base 64 encode look like:

```
{
  "iss": "scotch.io",
  "exp": 1300819380,
  "name": "Chris Sevilleja",
  "admin": true
```

}

And in base 64 format:
eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0

Signature

The third part of JWT is the signature which is made up of a hash of the header, payload and secret. The secret is the signature held by the server. This helps the server verify existing tokens and sign new ones. The signature is constructed as it shown below:

```
var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);

HMACSHA256(encodedString, 'secret');
```

And is base 64 format:
03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

The full JWT is:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm5hbWUiOiJDaHJpcyBTZXZpbGxlamEiLCJhZG1pbiI6dHJ1ZX0.03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

Extensible Authentication Protocol (EAP)

EAP is an authentication framework which supports multiple authentication methods. It runs directly over data link layers such as Point to Point Protocol (PPP) or IEEE 802, thus, it does not require an IP address. The advantage of the EAP architecture is its flexibility, as it is used to select a specific authentication mechanism, typically after the authenticator requests more information in order to determine the specific authentication method to be used. EAP permits the use of a backend authentication server, which may implement some or all the authentication methods, with the authenticator acting as a pass-through for some or all methods and peers. EAP supports more than forty different authentication methods, some of them are: EAP - Transport Layer Security (TLS), EAP - MD5, EAP - Tunneled Transport Layer Security (TTLS), EAP – Subscriber Identity Module (SIM), EAP – Authentication and Key Agreement (AKA), etc.

EAP is not a wire protocol; it only defines message formats. Hence, each protocol that uses EAP defines a way to encapsulate the EAP messages within the protocol ‘s messages. The most known protocols that encapsulate EAP are:

- IEEE 802.1x: EAP over IEEE 802 or EAP over LAN (EAPOL) is defined in IEEE 802.1x. When EAP is invoked by an 802.1x enabled Network Access Server (NAS),

modern EAP methods can provide a secure mechanism and negotiate a secure private key between the client and NAS that can then be used for a wireless encryption session.

- Protected Extensible Authentication Protocol (PEAP): PEAP is a protocol that encapsulates EAP within a TLS tunnel. It aims to correct deficiencies in EAP; EAP assumed a protected communication channel, such as that provided by physical security, so the facilities for protection of the EAP conversation were not provided. It was jointly developed by Microsoft, Cisco Systems and RSA Security.
- Point to Point (PPP): EAP was originally an authentication extension for the PPP and was created as an alternative to the Challenge Handshake Authentication Protocol (CHAP) and the Password Authentication Protocol (PAP). Eventually the above two protocols were incorporated into EAP.
- RADIUS and Diameter: These two are Authentication Authorization and Accounting (AAA) protocols, which encapsulate EAP messages. They are often used by NAS devices to facilitate IEEE 802.1x by forwarding EAP packets between IEEE 802.1x endpoints and AAA servers.

The EAP packets are defined in a binary format and their contents highly depend on the authentication scheme that is used each time. The EAP packet format is describing in the figure below. The fields are transmitted from left to right.

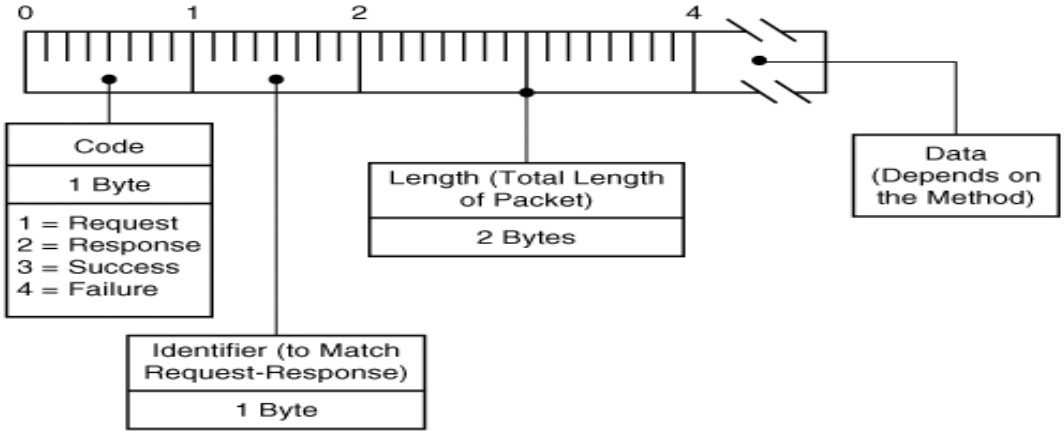


Figure 6: EAP Packet Format

Protected EAP (PEAP)

PEAP is an authentication method used over 802.1x. It utilizes server-side public key certificates in order to authenticate clients with server. The important thing and what makes PEAP authentication valuable and widely used combined with other EAP methods, is that it creates an encrypted TLS tunnel between the client and the authentication server. After the tunnel establishment, the credentials will be encrypted and protected against attacks like *packet sniffing* and *man-in-the-middle* when transferred.

Code	Identifier		Length
Type	Flags	Ver	TLS Message Length…
…TLS Message Length			TLS Data…. (EAP packets)

Figure 7: PEAP Packet Format

It requires only a server side PKI certificate to create a secure TLS tunnel to protect user authentication. PEAP is comprised of a two-part conversation:

In part 1, a TLS session is negotiated, with server authenticating to the client and optionally the client to the server. The negotiated key is then used to encrypt the rest of the conversation.

In part 2, within the TLS session, a complete EAP conversation is carried out, unless the part 1 provides a client authentication. It is very useful for WLANs as the users usually do not have a PKI certificate.

The whole PEAP procedure, including both phase 1 and phase 2, is described in the below figure.

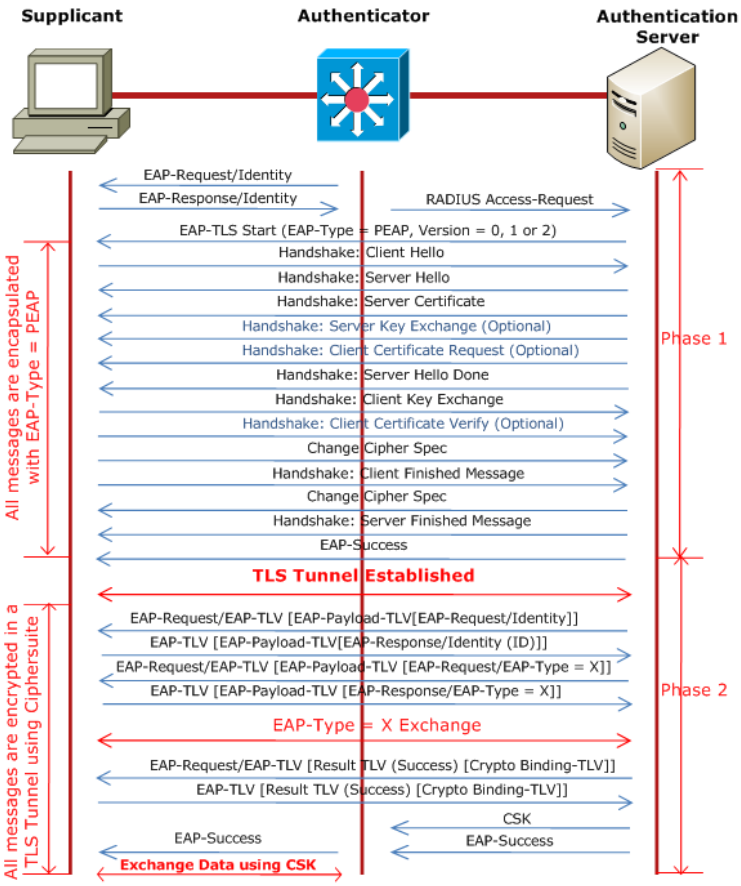


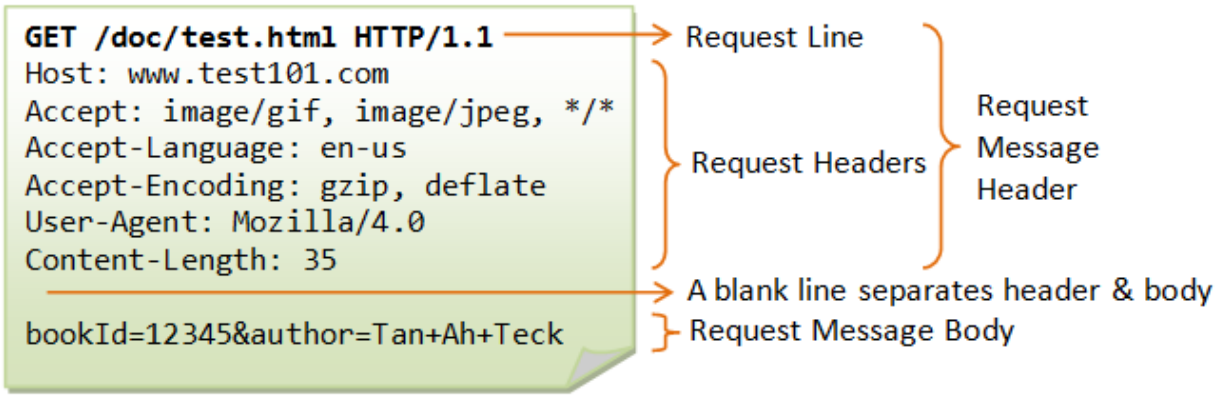
Figure 8: PEAP [20]

EAP-OIDC

HTTP

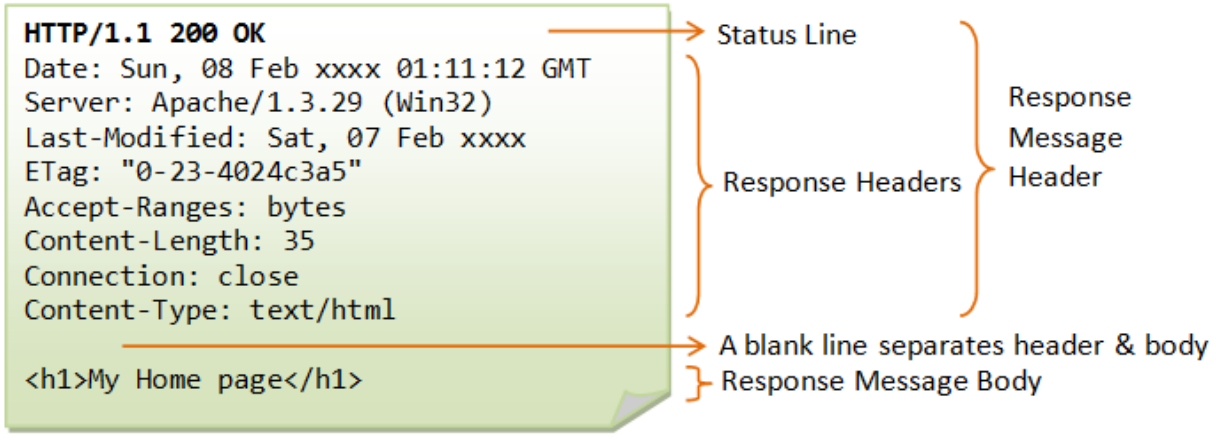
As we have mentioned before, OIDC consists of various Hypertext Transfer Protocol (**HTTP**) messages which are divided in two main categories [22]:

Requests from clients



- The Request Line begins by stating the method, followed by the Request-URI and the protocol version
- The request-header allows the client to pass more information about the request, and the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method invocation.
- The message-body of an HTTP message is used to carry the entity-body associated with the request or response.

Responses from Servers

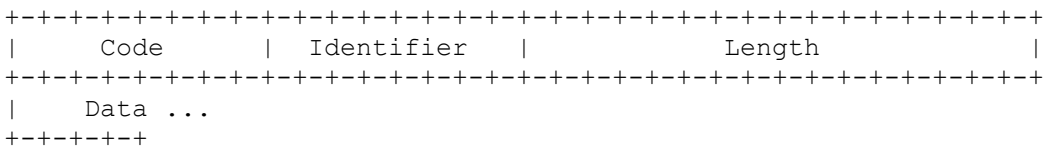


- The Status Line is the first line of a Response message, consisting of the protocol version followed by a numeric status code and its associated textual phrase.

- The response-header fields allow the server to pass additional information about the response which cannot be placed in the Status- Line. These header fields give information about the server and about further access to the resource identified by the Request-URI.

Encapsulating HTTP

These Application Layer messages cannot be transferred in this format directly to the authentication server, especially if they are to go over a wireless connection. To overcome this problem, we propose the encapsulation of HTTP within EAP packets. The EAP packets formatting is shown below.



The OIDC messages will be inserted at the “Data” area from the sender, and the receiver will have to reverse the process.

Remote Dial in User Service (RADIUS)

RADIUS is defined in RFC 2865 [7] and is an AAA protocol, which stands for: Authentication, Authorization and Accounting. It offers AAA services for users who connect and use a network service. At first it was developed as an access server authentication and accounting protocol and later became an IETF standard. It is also usually used to transport EAP packets between the Authenticator (Access Point) and the Authentication Server. The AAA functions of RADIUS protocol are described below:

- Authentication: The client sends an access request to the network, which contains user credentials or a user certificate. The authenticator convert this in RADIUS format and forwards it on to a RADIUS server. The RADIUS server checks its user database for a match and if it finds one the user is authenticated. The following messages are used: Access Reject, for rejection, Access Challenge, to ask for more information or Access Accept for acceptance.
- Authorization: The RADIUS server defines the terms of access for the user and more specific defines what are the user permissions in a network.
- Accounting: RADIUS accounting is enabled by the Authenticator if user access statistics and information are required. The Authenticator issues an Accounting Start Request to the RADIUS server. Following, in order to indicate information about the duration of the user session Interim Accounting Records may be sent. The Accounting is stopped when an Accounting Stop Record is sent to the server.

The authentication and authorization are defined in RFC 2865 while accounting is described by RFC 2866 [24].

Usually, the RADIUS protocol uses UDP ports 1812 for Authorization and 1813 for Accounting.

RADIUS Packet Format

The packet format of RADIUS is shown is the figure below. The fields are transmitted from left to right.

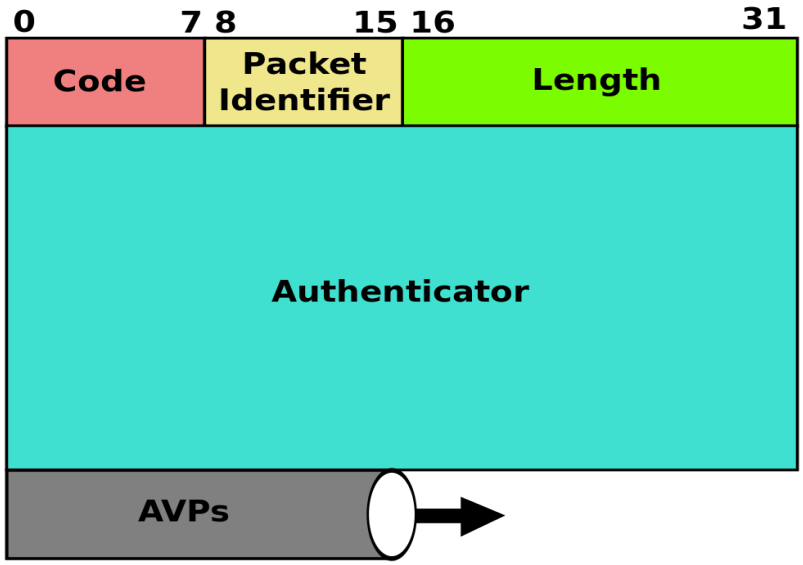


Figure 9: Radius Packet [25]

Code: This is 1 byte long and identifies various types of packets i.e. 1 for Access Request, 2 for Access-Accepts.

Packet Identifier: It is also 1 byte and aids in matching responses with requests.

Length: This is 2 bytes and specifies the length of the packet including code, identifier, length and authenticator.

Authenticator: This is 16 bytes and is used to authenticate the reply from RADIUS server, as well as in encrypting passwords.

Attribute Value Pairs (AVPs): The AVPs consist of Type, Length and Value. The Type is 1 byte and identifies various types of attributes. The Length, is also 1 byte an it describes the length of the attribute including Type. The Value can be either 0 or more bytes and contains information specific to attribute.

EAP Encapsulation in RADIUS

The transmission of the EAP packet from the user to the authentication server requires the encapsulation of EAP packets within RADIUS messages. More specifically, the whole EAP packet encapsulated into a RADIUS attribute [24].

- EAP-Message: This attribute encapsulates the EAP packets so as to allow the Network Access Server (NAS) to authenticate users via the EAP without having to understand the

EAP method it is passing through. The NAS places the EAP packets received from the authenticating user into one or more EAP-Message attributes and forwards them to the RADIUS server within an Access Request message. The EAP-Message attribute consists of: Type, which is 1 byte and is 79 for EAP-Message. Length, which is also 1 byte and is equal or bigger from 3 and String which contains an EAP Packet.

- Message-Authenticator: This attribute is used to authenticate and integrity protect Access Request from spoofing. It consists of: Type, it is 1 byte and is 80 for Message-Authenticator. Length, which is 18 and the String, which when present in an Access Request packet, is an HMAC-MD5 hash of the entire Access Request packet. For Access Challenge, Access Accept and Access Reject packets the Message Authenticator is the HMAC-MD5 of the Type, Identifier, Length, Request Authenticator, Attributes.

OpenAM

While doing research for this Thesis, we wanted to see a real-life implementation of OIDC, in order to observe the way it works and get a better understanding. Hence, we installed and configured the OpenAM in a Linux Ubuntu. OpenAM is an open source access management platform sponsored by ForgeRock. Formerly, it originated as OpenSSO and was created by Sun Microsystems, but now it is owned by Oracle Corporation. The features that are supported by OpenAM include among others Authentication, Authorization, Adaptive risk authentication, Federation, SSO.

We chose to use the OpenAM to act as a Relying Party and to use our Google accounts to authenticate in it. Moreover, we created a project in Google APIs page in order to retrieve the Client ID and Client Secret. The procedure for retrieving the aforementioned credentials is beyond the scope of this chapter, as it depends on each IdP.

OpenAM supports more than 20 authentication methods. It is also offers a user friendly environment, in which is very easy to configure social authentication. As it shown in the image below, the user only has to go in **Dashboard** tab and click in the **Configure Social Authentication** button. After that the user is prompted to choose the social account that they

want to use. As we mentioned before, we want to use a Google account, so we click the **Configure Google Authentication**.

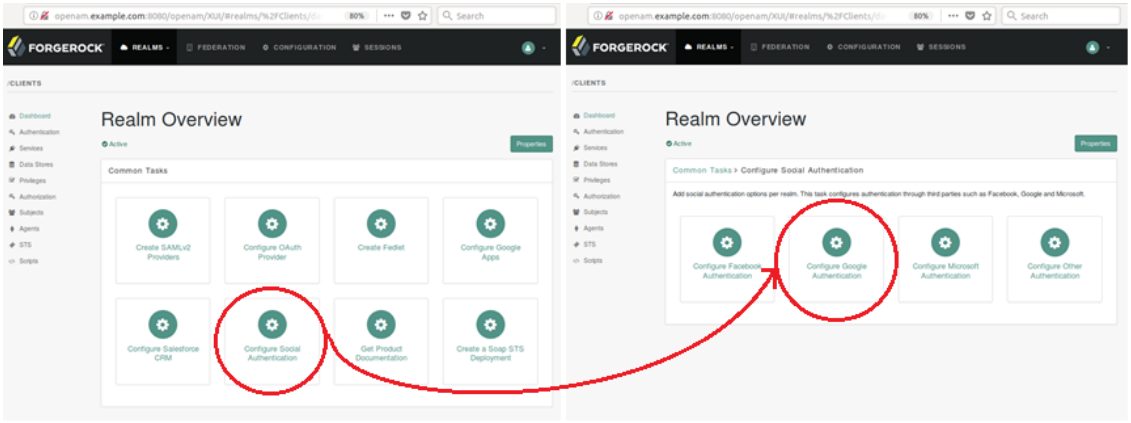


Figure 10:OpenAM Social Authentication & Options

In this figure the user is prompted to give the credentials that were retrieved from the IdP 's website. In our case, we use the Client ID and Client Secret, which we retrieved from Google.

Figure 11: Configure Client Credentials

To edit the Google Social Authentication, we have to go to **Authentication => Modules** tab as it shown in the figures below.

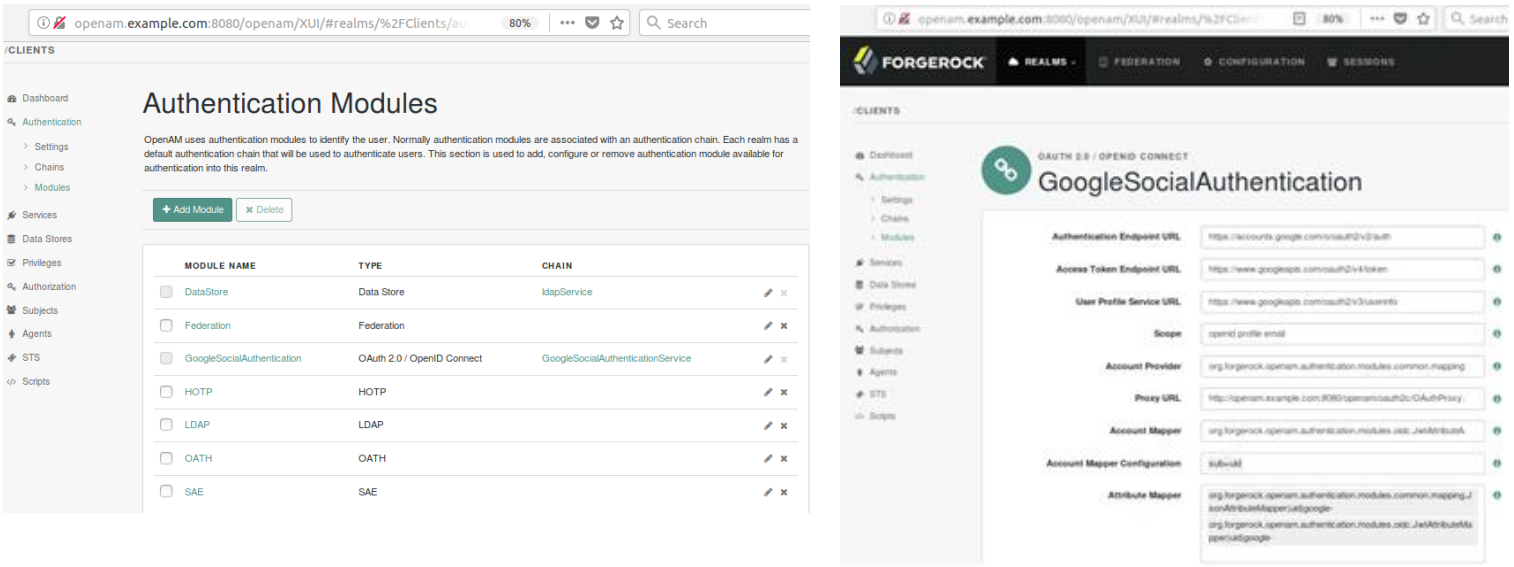


Figure 12:OpenAM Authentication Modules & Google Social Authentication Edit Page

In order to authenticate in OpenAM with Google account the user should click the G+ button. Then he is redirected in a login page that is offered by Google to fill their account credentials.

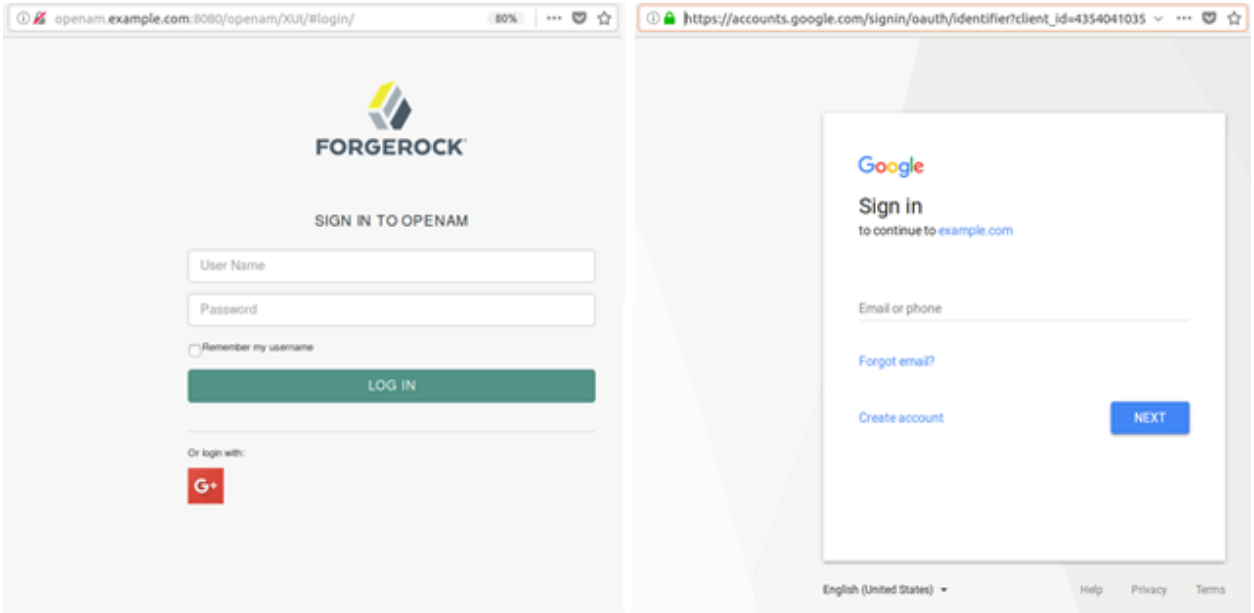


Figure 13: OpenAM Login Page & Google 's Login Page

If the user credentials are correct they are redirected back in OpenAM's user profile page as it presented below.

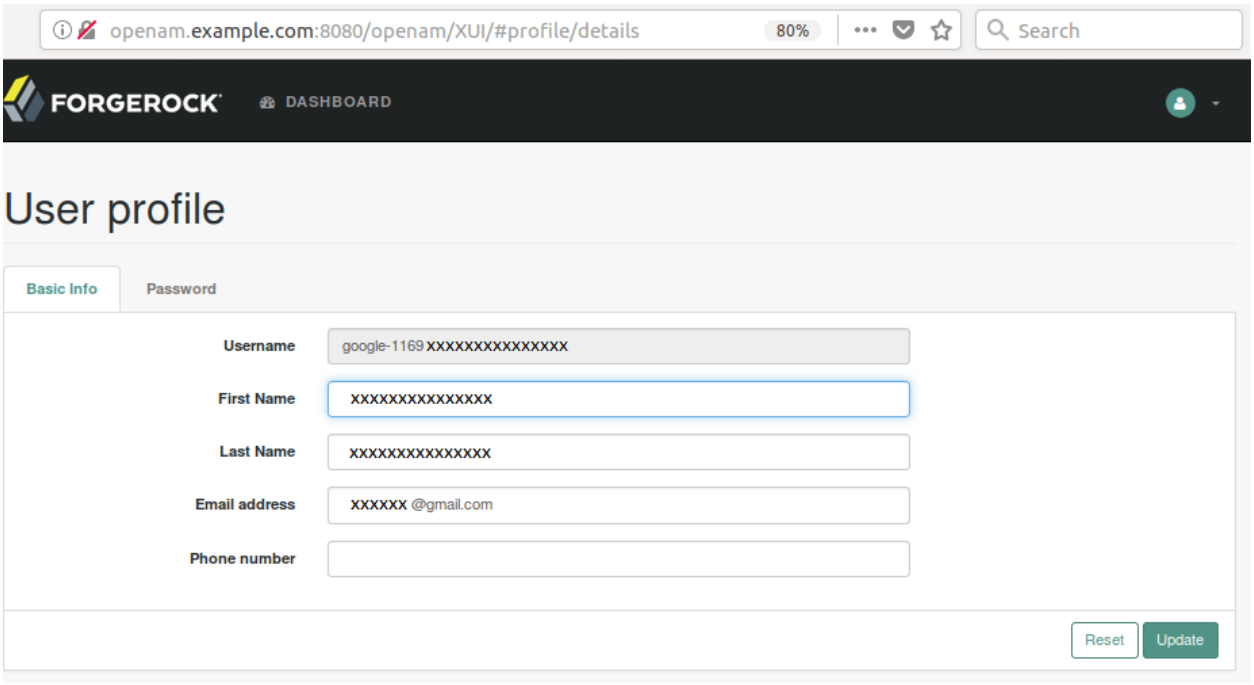


Figure 14:OpenAM Profile Page

Related Work

Eduroam

Eduroam is a solution very similar to the one proposed in this thesis. It offers the academic community the ability to connect to the Wi-Fi hotspots that belong to Eduroam, using their academic credentials. The network basically consists of numerous RADIUS proxy servers.

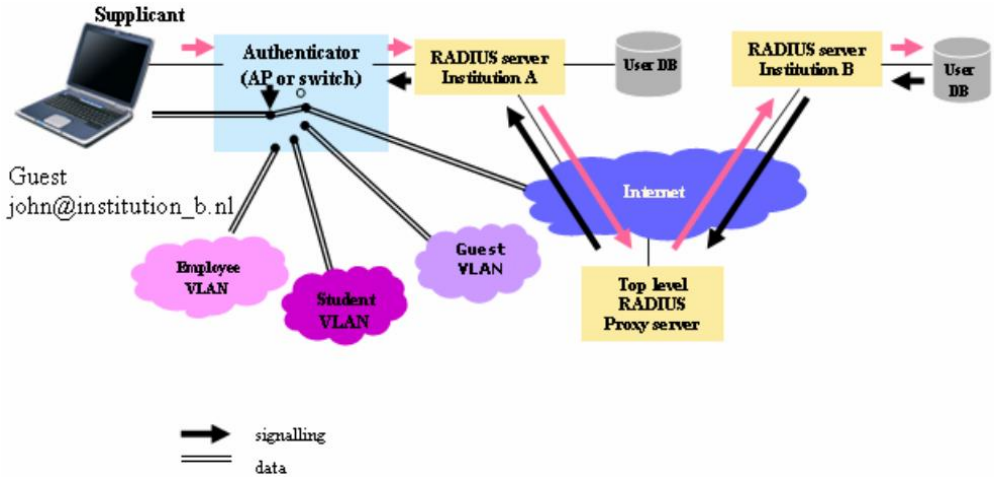


Figure 15: Eduroam Infrastructure [27]

When a user requests authentication, the user's realm determines where the request is routed to. The realm - i.e. @unipi.gr - is derived from the organization's domain name. [26] Every institution that wants to participate in Eduroam connects its institutional RADIUS server to the federation level RADIUS (FLR) server of the country where the institution is located. Depending on the realm each user belongs to, the proxies will rely the message to the corresponding authentication server to verify the credentials that were provided.

Captive Portal

Captive portal is a web page, which is displayed to newly connected users before they are granted broader access to network resources. Usually, it is stored either at the gateway or on a web server hosting that page. Websites or TCP ports can be white-listed so that the user would not have to interact with the captive portal in order to use them. Typically, it is used in airports, hotels and other venues that offer free Wi-Fi hot spots for Internet users. Captive portals have several uses. They are primarily used in open WLAN networks, where a welcome message is shown, informing the users about the terms and conditions of access. Administrators tend to do this to avoid any legal responsibility, as the users take responsibility for their actions. Sometimes, captive portals are used for marketing and commercial purposes. The user is unable to browse freely – "captive" until they accept the terms and conditions. This allows the provider of this service to send or display advertisements to users who connect to the Wi-Fi access point. This type of service is also known as social Wi-Fi, as they may ask for a social network or email account to login. Over the past few years, many companies offer marketing centered around Wi-Fi data collection.

Moreover, with captive portal, unauthenticated users attempting to access the network are redirected to a captive portal web page. Users obtain IP but the access to network resources is restricted until they are authenticated via a browser-based login. Captive portal authenticates users at Layer 3 (network layer), the encryption is typically done at the level of the browser using the HTTPS protocol.

This method has a browser-based login for the users' authentication and the exchange of social login credentials is based on OAuth 2.0 (Open standard for Authorization). The Universal Access Method (UAM) protocol is used, which allows a user to access a Wi-Fi network by using a browser.

A typical authentication flow using captive portal proceeds as follows:

1) User associates to the Wi-Fi network,
2) The Wi-Fi AP, using UAM, redirects the user browser to the captive portal web page,
3) The user gives the social account credentials. The login window is a secure redirect back to the social account login page.
4) If the authentication credentials the user presents are valid, the captive portal server will receive a success message from the social account server.
5) The captive portal server creates temporary RADIUS credentials on the RADIUS server and sends a message to the AP which includes the temporary credentials.

6) The AP sends an authentication request using this credentials to the RADIUS server,
7) The RADIUS server responds with a success message and a set of RADIUS attributes that control the user session parameters.

WPA2 / 802.1X Authentication

During our research about methods which support Wi-Fi authentication, using social media or email accounts, besides the captive portal we found some enterprise solutions [17] [18] [19], that achieve the Wi-Fi authentication with Google credentials, using protocols like WPA2 and 802.1x. These methods achieve better security levels than the captive portal, but come with a cost as they are not open source solutions. With WPA2/802.1x, authentication happens before an IP address is granted on a user and allowed on the network. This method works at Layer 2 (data link layer). In this case, the wireless client is authenticated, the encryption key is derived and the Layer 2 wireless connection between the client and the access point is encrypted. This protects against attacks at upper layers by denying access before a rogue user ever gets on the network. In a wireless network, the 802.1x authentication occurs after the end user has associated to an AP using 802.11 association method.

This is accomplished by using EAP TTLS protocol to set up a secure tunnel between end user and RADIUS server, and Password Authentication Protocol (PAP) to pass the credentials of the social media account from the end user to the RADIUS server and over to Google for authentication and authorization. The authentication flow proceeds as follows:

- EAP TTLS/PAP first authenticates the connection between the Wi-Fi AP and the RADIUS server and sets up a trusted secure TLS tunnel between the AP and the RADIUS server.
- Once the TLS tunnel is established between the AP and the RADIUS server, the AP will send authentication credentials as PAP protocol messages.

Motivation

Everything we mentioned in Introduction about the average person and that he has 27 discrete logins combined with the fact that there is no authentication mechanism offering social media or email account authentication in a Wi-Fi network (without compromising security), lead us to design a methodology that uses the EAP among with OIDC. There are some enterprise solutions that offer social media and email login, as providers are already Identity Providers and they already use Single Sign On (SSO) services and OIDC protocol. The main difference in our approach is the service, because it is not a web application, but a Wi-Fi network. On the one hand OIDC protocol is used to provide such an authentication method, but is based on web applications. On the other hand, EAP framework is used for almost one and a half decade and supports multiple methods for Wi-Fi network authentication. So, we came up with the idea of connecting these two protocols in order to provide one strong and easy to implement authentication protocol. Moreover, eduroam uses the same network components with our approach, like RADIUS servers and protocols like 802.1x with an EAP method and works very

good. However, offering a public Wi-Fi in which everyone will use their own credential to access it, not only will make it more secure, but also user friendlier, as the user will not have to look for passwords in order to access a network.

Contribution

As mentioned before, our approach combines two well-known protocols in order to create one new authentication protocol. This new protocol will provide the opportunity to add security measures in public Wi-Fi networks without losing the ease of use. The credentials of the users will not be accessed by the Wi-Fi network provider, the only one that will have access to the credentials will be the identity provider. In order to achieve that and keep the credentials secret from eavesdropping the EAP packets should be transferred through an encrypted channel [10]. In addition, OIDC is based on HTTP protocol, hence our approach will open the road for another new EAP method which will be based on HTTP. The implementation of our methodology is going to be neither difficult, nor particularly expensive, as EAP and OIDC protocols are well defined. Our proposed method offers layer two (data link layer) security as the EAP runs directly over data link layers without requiring IP address, unlike captive portal, which offers an application layer security since users first obtain an IP address and then the captive portal limits users' network connection. What is more, the combination of EAP and OIDC will lead to a new authentication protocol which will be ease to adopt by the SPs as it is based on EAP.

Analysis as a Use Case

The Scenario

EAP methods are mostly used in wireless networks, depending on the owner's requirements and security policies. The idea of someone connecting to Wi-Fi with the e-mail credentials is best suited for an environment where both security and access control are valued. Having that in mind, we choose to present the connection to a corporate wireless network to present our approach. Not only does this help us to see if our solution could be used as real use case scenario, but also it helps the reader to deeply understand our method.

More specifically, we consider a company that wants to implement a wireless network which will allow only its employees to connect easily from any device they own, without having to remember/enter a new password. It is common practice that each employee at a company gets an e-mail address (e.g. someone@company.com) accompanied by a password; the same credentials could be used to login to the wireless network. The Access Point could be anywhere as long as there is a communication channel with the provider's authorization server. For example, an employee based in Athens, could travel to London and when in the company's premises, there would be no need for new credentials (or the use of certifications that are difficult for users to install and update) and the employee could connect to the network instantly.

In order to accomplish that, we will use PEAP combined with EAP-OIDC method to transfer OIDC messages. Next, we will see the authentication phases that will follow a user's push of the "Connect" button on a device:

Phase 1

First of all, a secure TLS tunnel will be established as shown below:

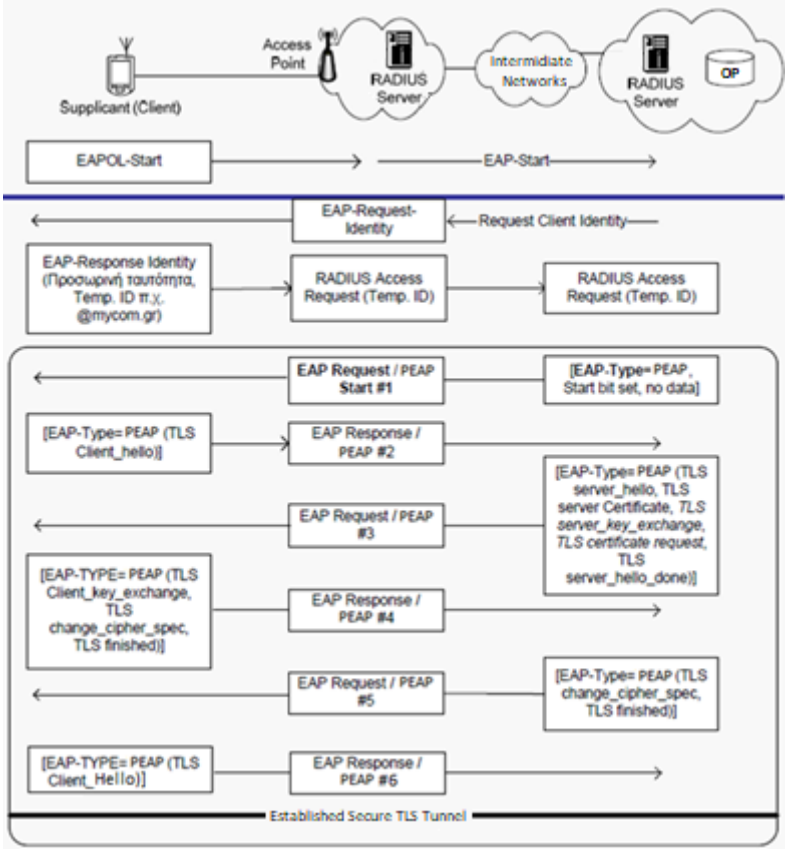


Figure 16: TLS Tunnel Establishment

Phase 2

Now that the communication channel is secured, the EAP-OIDC authentication will commence, following the principles aforementioned in OIDC Authorization Code Flow. After the user fills in his credentials to the form that will be presented to him, the OIDC messages will be encapsulated within EAP packets and transferred via the TLS channel, authenticating the end user to the AP.

PEAP with EAP-OIDC

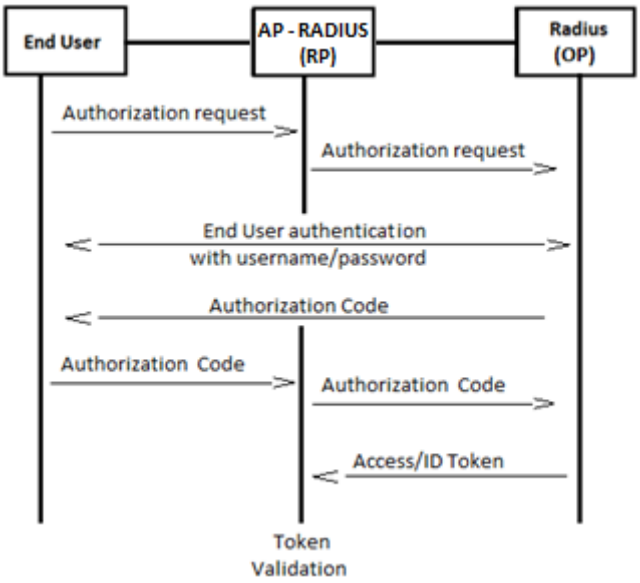


Figure 17: Overview of PEAP with OIDC

Essentially, the following steps will be completed:

1. Service Provider (AP – RADIUS) sends an authorization request to the appropriate OIDC Provider – company Server in this case, which is able to validate the credentials as it already has them
2. End user gets authenticated and receives an Authorization Code
3. SP receives the Authorization Code, which in turn gets sent back to the OP
4. SP is given an Access and an ID Token in exchange for that Authorization Code
5. If the Access Token gets validated successfully by the SP, then the end user will be given access to the Wireless Network

This process can be simplified if the user has saved the forms with his credentials – so he will not have to type them in every time he tries to connect – and the company's wireless network name is the same worldwide; if that was the case, the connection process would be completely automated. The downside of this practice is that if the device is misplaced, third parties (including competitors and criminals) could easily access company data and resources if there are no other measures to protect logical access to the device. The consequences could be catastrophic for the company, but this can be prevented by placing suitable corporate Information Security Policies and making sure that the employees are adequately trained and follow the practices described in the policies.

Benefits/Advantages

One of the best things about EAP is that it runs directly over data link layer and by extension it does not need an IP address to work. This means that the user obtains an IP address after a successful authentication. The alternative no cost and open source solution which may offer

authentication with social media or email accounts is implemented with captive portal which may be easier to implement, but does not offer such a high level of security. The methodology proposed in this Thesis, will be salutary for public Wi-Fi networks, as it enhanced security. For open and unprotected networks, it provides a password without the need to be remembered by the users, since they use credentials that are already known to them. Networks which are protected with a pre-shared key are not secure enough as every device connected to the AP use the same "shared encryption" connection, so eavesdropping the traffic is possible. The use of an EAP authentication method requires the AP to be connected with a RADIUS server, using RADIUS protocol, which encrypts uniquely the session between the user and AP. Moreover, another benefit for public Wi-Fi networks with our solution is that RADIUS server could be used for access control (mainly in corporate environments) without having to create new accounts. What is more, we propose the use of PEAP, which adds an extra security level as it is established a TLS tunnel and the user's credentials are transferred within this tunnel. In addition, as we already mention in the SP is implemented a proxy RADIUS server, which should transfer the credentials from the user to the OP. With the TLS tunnel the SP will not have access in user 's credentials, hence our methodology respects the user's privacy.

Possible Drawbacks

Our proposed methodology in order to be applied requires the use of RADIUS protocol for the transmission of the EAP packets between the user and the authentication server which is located in the OP. More specifically, the EAP data chunks are inserted in RADIUS protocol messages. The way EAP is transported over RADIUS is described in [23]. Hence, the drawbacks that occur from our approach is the implementation of RADIUS servers, that is required in order to achieve the EAP-OIDC authentication. The use of an EAP method for authentication in a WLAN network adds the limitation that the AP should communicate with a RADIUS server. In our methodology, as we mentioned before, the RADIUS server in the SP should be implemented as a proxy for transferring the credentials from the user to the OP. This means that besides the AP, SPs should also have a RADIUS Server configured as a proxy which would transfer the EAP packets to another RADIUS server in the OP. From the OP scope, a RADIUS server should be implemented to receive the messages from the proxy and transfer the OIDC messages in the Authorization server, as the use of the RADIUS protocol cannot exist without the RADIUS server. This will add an extra component in the network infrastructure, although nowadays the implementation of RADIUS servers is neither time consuming, nor expensive. Another drawback that occurs with our proposed methodology is that the overhead will be significant greater than with the already implemented authentication methods. At this time, we cannot measure the difference, as our methodology is not implemented yet. Hence, this is a theoretical assumption which is based in the multiple encapsulation layers that come of from our method.

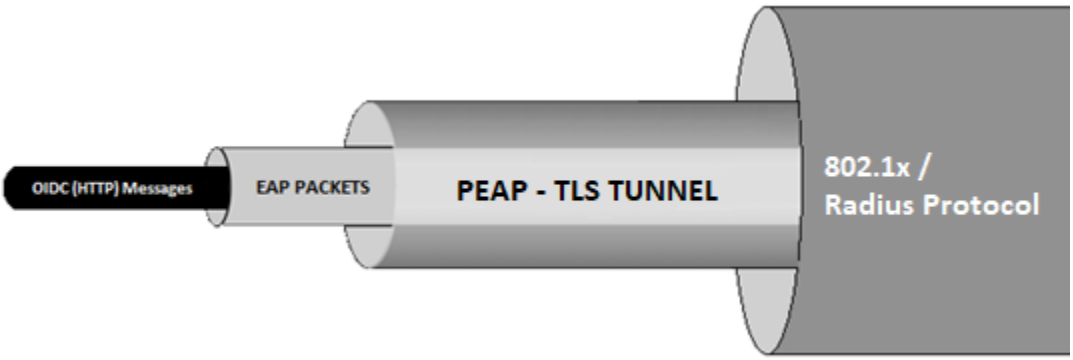


Figure 18: Encapsulation Overview

As it presents in the figure above, there is a quadruple encapsulation. In order to prepare a message like this, significantly more resources will be utilized, compared to i.e. a common connection with WPA2.

Conclusion

In the fast-paced days we live in, everybody wants to do everything as fast and simple as possible. Connecting to wireless networks using OpenID Connect, and subsequently taking advantage of existing credentials and not creating new ones, is a solution mainly addressed to businesses or universities. It can benefit employees by providing a simple and easy way to connect in the organization's network from any location around the world (i.e. different cities/countries), or offer customers an equally simple and secure method to enjoy free Wi-Fi from their devices, along with the main services provided to them by the company.

The implementation of this solution mostly rests on successfully encapsulating HTTP messages in EAP packets, reversing the procedure and providers setting up RADIUS servers to handle the EAP packets they will receive. This is what needs to get done in order to have a functional PEAP-OIDC authentication on wireless networks.

References

[1] Extensible Authentication Protocol RFC 374, 2004.
[2] https://backstage.forgerock.com/docs/am/5/oidc1-guide/
[3] Captive Portal - https://en.wikipedia.org/wiki/Captive_portal
[4] Facebook Wi-Fi - https://www.facebook.com/business/facebook-wifi
[5] WPA and WPA2 https://www.wi-fi.org/certification/programs
[6] http://www.ieee802.org/1/pages/802.1x-2004.html
[7] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote authentication dial in user service (RADIUS). RFC 2865, 2000
[8] OpenID Connect - http://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth
[9] D. Hardt, Ed. Microsoft. The OAuth 2.0 Authorization Framework. RFC 6749, 2012
[10] Ashwin Parker, Dan Simon, Microsoft, Glen Zorn, Cisco, S. Josefsson, Extundo. Protected EAP Protocol

[11] Social WiFi - https://socialwifi.com/
[12] http://cloudessa.com/solutions/using-google-apps-for-wifi-authentication/
[13] https://www.globalreachtech.com/google-apps-wifi-authentication/
[14] https://www.buzzfeed.com/josephbernstein/survey-says-people-have-way-too-many-passwords-to-remember?utm_term=.akQ5wnv2Y#.mlqAKopnq
[15] Eduroam - https://www.eduroam.org/
[16] K. Wierenga, L. Florio, Eduroam, providing mobility for roaming users, 2005
[17] http://cloudessa.com/solutions/using-google-apps-for-wifi-authentication/
[18] https://www.globalreachtech.com/google-apps-wifi-authentication/
[19] https://jumpcloud.com/blog/g-suite-credentials-wifi-authentication/
[20] https://sites.google.com/site/amitsciscozone/home/switching/peap---protected-eap-protocol
[21] https://changchen.me/blog/20170321/http-protocal/
[22] https://www.ietf.org/rfc/rfc2616.txt
[23] B. Aboba Microsoft, P. Calhoun Airespace. RADIUS (Remote Authentication Dial in User Service) Support For Extensible Authentication Protocol (EAP), RFC 3579, 2003.
[24] C. Rigney Livingston. RADIUS Accounting, RFC 2866, 2000.
[25] https://freeradius.org/
[26] https://www.eduroam.org/about/
[27] Licia Florio, Klaas Wierenga Eduroam, providing mobility for roaming users