VPN User Authentication Using Centralized Identity Providers

Duarte Mortágua
IEETA, University of Aveiro
Aveiro, Portugal
email: duarte.ntm@ua.pt

André Zúquete, Paulo Salvador
DETI / IEETA, University of Aveiro
Aveiro, Portugal
email: {andre.zuquete,salvador}@ua.pt
0000-0002-9745-4361, 0000-0001-6832-9417

***Abstract*—The online access to an always growing set of services requires users to manage credentials to identify themselves to all of them. The reduce this burden on users, centralized authentication systems, ordinarily known as Identity Providers (IdPs), and Single Sign-On (SSO) protocols where developed and are often deployed. IdPs and SSO were mainly developed for Web-based interactions, first in the scope of a set of federated services belonging to one organization, later on wider scopes, such as for virtually everyone (e.g., Google or Facebook users) or for all citizens of a given country. The Portuguese national IdP, Autenticação.gov, is an example of this later case. Today, many adhering services, from both the public and the private sectors, enable users to authenticate themselves using the functionalities provided by Autenticação.gov. However, the use of this IdP, as well as of similar ones, is mostly limited to Web applications. The goal of this paper was to study the integration of IdP services with Virtual Private Network (VPN) setup processes, namely for the authentication of VPN users. To this end, we used a recent VPN technology, WireGuard, which became popular amongst vendors due to its speed, simplicity and adoption by the kernels of the mainstream operating systems. We propose a method for a WireGuard-based VPN client to connect to a VPN server and negotiate cryptographic keys associated to a user authenticated by a centralized, OAuth 2.0-enabled IdP. We implemented a VPN server that enables users to use two different IdPs, namely Google Identity and Autenticação.gov; they both support the OAuth 2.0, but in different ways.**

***Index Terms*—Identity Providers, Authentication, OAuth 2.0, VPN, WireGuard**

I. INTRODUCTION

Services provided by companies or public sector departments often require people to register themselves, i.e., to create an account. Such registration usually involves the provisioning of users' authentication credentials (usually a passphrase) and a recovery mechanism (usually an e-mail address or, more recently, a phone number). Furthermore, and normally more complex to validate, users associate extra identity data to their account that may be useful in the future (e.g., a P.O. box address, a payment method, etc.).

Centralized Identity Providers (IdPs) appeared to reduce the users' burden regarding account management. They permitted to evolve from a so-called silo approach (where services do not share accounts) to accounts that can be shared by a set of federated services. Those services, often called Relying Parties (RPs), trust on the user authentication implemented by an IdP, and sometimes they can even enforce the use of specific approaches, by specifying Level of Assurance (LoA) indications. Furthermore, the RPs also receive, upon a user authentication, a set of user identification attributes which they assume that are accurate. Such accuracy is the responsibility of a back-office service used by an IdP, the Identity Manager (IdM).

Single Sign-On (SSO) is a concept that leverages centralized IdPs. Besides the centralization of the authentication in an IdP, SSO also enables users to remain authenticated during a time lapse, defined by the IdP (an authenticated session). Consequently, during that time they can access any federated RP without having to be authenticated for each of them.

IdPs and SSO were first explored in the context of Web interactions through the use of messages formatted with Secure Assertion Markup Language (SAML) [14] exchanged between an IdP and an RP through HTTP-based protocols, such as the Web Browser SSO Profile [13]. More recently, IdPs and RPs started to use OAuth 2.0, a protocol conceived to implement access control delegation, to allow RPs to access user identity resources maintained by an IdP. Nowadays, popular Internet services, such as Google and Facebook, which authenticate millions of people, an keep some relevant user identity attributes in their accounts, are often used as IdPs.

In order to facilitate the online identification of people in their interaction with services provided by the private or public sectors, several countries deployed an IdP for their citizens. This is the case of Autenticação.gov, created and maintained by the Portuguese state. This IdP enables citizens to authenticate themselves using two alternative methods, both implementing a two-factor authentication: a personal electronic identification device (Cartão de Cidadão, an eID crypto token) with a secret PIN or a combination of a secret PIN and a mobile phone number or e-mail address (Chave Móvel Digital). Other European countries followed a similar approach, for instance the ID Austria [3] or Cla@ve [9] in Spain.

This growing use of centralized IdPs for authenticating users and providing identification attributes about them to RPs happens mainly in the context of Web-based interactions, and considering that users use Web browsers to access the services provided by RPs. In this paper, we describe how we can explore an IdP for user authentication during a VPN setup, an action that became more frequent upon the recent Covid-19

Copyright (c) IARIA, 2023. ISBN: 978-1-68558-052-0

pandemic. We chose to use a VPN technology, WireGuard, that uses as host (or user) authentication paradigm a set of public keys that must be pre-shared between VPN client and server. Then, we designed and implemented a protocol, involving a user browser and an IdP, which enables a VPN server to receive a trustworthy binding between a WireGuard public key and a user identity attribute. The provisioning of the identity attribute requires a user authentication by the IdP within the VPN setup protocol. For IdP services, we used Google Identity and Autenticação.gov. They both support OAuth 2.0, in different ways, to allow RPs (our VPN servers) to fetch a controlled set of identity attributes from the users they authenticate. A proof of concept implementation of the VPN client and server was successfully tested with those IdPs.

It is worth noting that we are not proposing a new authentication mechanism. We are proposing to benefit from the authentication mechanisms already explored by IdPs, and the subsequent provisioning of identity attributes associated to the authenticated person, during the setup of VPNs.

This paper is structured as follows. In Section II, we briefly describe the OAuth 2.0 details that are relevant to understand its use by IdPs and we detail how Google Identity and Autenticação.gov use it. In Section III, we explain the setup of a WireGuard VPN. In Section IV we describe the architecture of our solution. In Section V, we describe the implementation supporting two IdPs, Google Identity and Autenticação.gov. In Section VI, we discuss the security and usability of the final system. In Section VII, we present some related work. Finally, in Section VIII, we conclude the paper.

II. OAUTH 2.0 IN THE CONTEXT OF IDPS

OAuth 2.0 is a protocol that can be used to authorize the access to protected resources in many different way. Thus, it can be explored differently by each IdP. In order to understand the integration of Autenticação.gov and Google Identity with our VPN setup process, it is therefore necessary to understand how these IdPs explore it in the context of users' identification. We start by an initial presentation of OAuth 2.0 concepts, and then we show how Autenticação.gov and Google Identity use it for user identification.

A. OAuth 2.0 concepts

OAuth 2.0 [10] enables an application (**client**) to access resources owned by a person (**resource owner**) kept by a **resource server**. In a nutshell, the client leads the resource owner (through their browser) to interact with the resource server in order to get an **authorization grant** to the client for accessing a set of resources. This interaction is conducted by the **authorization server** component of the resource server, which requires the authentication of the resource owner, shows the client identification and asks for permission to grant an access authorization to the set of resources listed by the client. This interaction ends successfully with the upload of an authorization grant to the client, which then uses it to get an **access token** from the authorization server. Finally, the client uses the access token as a bearer token to access the intended resources, kept by the resource server.

In the context of the centralized provisioning of personal identity attributes to federated services (RPs), the client is the RP, the resource owner is a person known by the IdP, and which the IdP knows how to authenticate, and the resource and authentication servers are parts of the IdP.

OAuth 2.0 was conceived for providing authorizations for clients wishing to access any kind of resource kept by a resource server. Thus, identity attributes are just a subset of those resources. There is an identity layer, called OpenID Connect (OIDC), that operates over OAuth 2.0, which uses ID tokens as resources. The knowledge of this layer is not fundamental to understand our system, and, in fact, we did not use it, because Autenticação.gov does not use it and Google Identity can be used without it. Consequently, we are not going to detail how it works.

B. OAuth 2.0 grant types

An OAuth 2.0 authorization grant can be obtained with 4 different approaches, which also define different interaction flows. Two of them, **resource owner password credentials** and **client credentials** grants, are not relevant, because they are meant to be used in special cases (when the client belongs to the resource owner and when the client is the resource owner, respectively) that are not suitable for our scenario.

The two grants that are of interest are **authorization code** and **implicit**. In the first case, the client receives an authorization code grant that it later uses to fetch an access token for accessing the resources of interest. The provisioning of the access token requires the client authentication by the resource server. In the second case, the client receives directly the access token. This approach is intended to be used by clients that are not meant to be authenticated by the resource server.

C. Registration of clients

The use of OAuth 2.0 implies a previous registration of the client in a resource server of interest. The registration requires the provisioning of the following items:

- Client type, either **confidential** or **public**. A confidential client is able to protect from disclosure a secret that can be used to get authenticated by the resource server (or its authorization server). A public client, on the other hand, cannot ensure the protection of such secret.
  In our case, the client will be an instance of a VPN server. It can be confidential, but in that case it requires a registration of each VPN server instance in the resource server. Or it can be public, if no secret is used or if the same secret is embedded in the code of all VPN server instances.
- Client identifier. This is a value that uniquely identifies the client in the scope of the resource server. It is provided by the later upon accepting the registration.
- Client authentication credentials. The resource server defines the alternatives (usually a password). Public clients may be required to make this registration, although not

enforcing a trustworthy identification (since they cannot protect the secrecy of the credentials).
- Redirection endpoint. This is a Universal Resource Identifier (URI) that is used by the resource server (or, more specifically, by its authorization server) to send the authorization grant to the client.
  In our case, we have several alternatives for defining this URI. We could have a URI per VPN server instance, but that would require a registration on each VPN server setup. Alternatively, since the URI is used in a communication initiated by a user browser (through HTTP redirection), the URI can contain an IP address that represents the browser host (e.g., 127.0.0.1). This solution is more appropriate for an exploitation scenario where there is a single registration for all VPN server instances (created by the VPN developers).
- Client identification items, such as application name, website, description, logo image, etc. Those items will allow users to recognize the client when it requires their identity attributes.

D. Autenticação.gov

Autenticação.gov uses the implicit grant flow with public clients without a shared secret. The registration of an OAuth 2.0 client in Autenticação.gov requires a manual agreement between the two parties, and the requester is naturally assumed to be an organization with an online portal. The registration includes 4 items: request issuer (portal URL), organization (the client identification to be presented to users), a contact e-mail (for technical issues) and a redirection endpoint domain (an IP address or a DNS domain). From this agreement results a client ID, which the client uses to identify itself in Autenticação.gov.

According to the public technical documentation of Autenticação.gov [1], the client needs to follow 3 steps regarding the user authentication and attribute provisioning:

1) Obtain an access token upon a successful user authentication by Autenticação.gov. This implies sending, to be presented to the user, the set of identity attributes the client wants to receive (scope). This step is initiated with a GET redirection of the user browser from the client to a specific Autenticação.gov authorization endpoint (see Table I). Upon a successful user authentication and authorization (to convey the indicated attributes), the browser is redirected to the client's redirection endpoint with an access token as a URL fragment.
   The client can specify how it wants the IdP to authenticate the user by means of an extra field in the initial request (`authentication_level`). If absent, the IdP will present to the user any available method.
2) Obtain an identifier of the authentication process (`authenticationContextId`) by presenting the access token and an optional subset of the attributes in the identification scope to Autenticação.gov with a JSON body.
3) Obtain the needed attributes by presenting the access token and the `authenticationContextId` to Autenticação.gov.

In our work we used a pre-production instance of Autenticação.gov. However, it is essentially a mirror of the production instance.

E. Google Identity

Unlike Autenticação.gov, the usage of the Google Identity IdP can be configured in an automated way through Google Cloud. This service allows the creation of Google Cloud projects, which may expose APIs and services to users. One of those services is Credentials, which allows the creation of OAuth 2.0 clients that may interact with Google APIs. The OAuth 2.0 consent screen presented to the users must also be configured, alongside the needed Google APIs that the OAuth 2.0 client may access, i.e., the OAuth 2.0 scopes.

The Credentials service allows the creation of multiple types of OAuth 2.0 clients [8], depending on the nature of the client application (Web App, JavaScript App, mobile App, etc.). In our case, the Desktop App was chosen, due to the fact that it allows the redirection of the Google's authorization server responses to a localhost-based redirection URI, with an arbitrary port, i.e., our VPN client running on the user's machine.

When a Credential is created, it generates a Client ID and a Client Secret, that can then be used by the client to implement the OAuth 2.0 flow. As opposed to Autenticação.gov, which uses the implicit grant flow, Google Identity uses the authorization code grant flow, which imposes client authentication towards the IdP.

In this case, there are also 3 steps for obtaining the attributes of a user:

1) Obtain an authorization code upon successful user authentication by Google Identity. Similar to the Autenticação.gov first step, this step implies making a GET request to Google's authorization endpoint (see Table I) that carries the OAuth 2.0 mandatory authorization parameters, such as the Client ID, the scopes and the redirection URI. After a successful user authentication, the user browser is redirected to the client's redirection endpoint with an authorization code in the URL query parameters.
2) Obtain an access token upon successful client authentication. In our case, this means the VPN server authentication with its Client ID and Secret, alongside with the authorization code. For this, one needs to use the Google Identity OAuth 2.0 token endpoint with URL encoded body).
3) Obtain the needed attributes upon presenting the access token to a Google API endpoint which the token is allowed to access, which is one of the token's scopes with the `Authorization` header as `Bearer` followed by the access token).

In the case of Google Identity, it was possible to publish the registered OAuth 2.0 client in production, which means

TABLE I
ENDPOINTS AND ACCESS METHODS FOR AUTENTICAÇÃO.GOV (TOP) AND GOOGLE IDENTITY (BOTTOM)

Endpoint	HTTP method and URL	
Authorization	GET	https://autenticacao.gov.pt/oauth/askauthorization?redirect_uri=...&client_id=...&response_type=token&scope=...
Client's redirection	GET	redirect_uri#token_type=bearer&expires_in=...&access_token=...
Authentication identifier request	POST	https://autenticacao.gov.pt/oauthresourceserver/api/AttributeManager
Attribute request	GET	https://autenticacao.gov.pt/oauthresourceserver/api/AttributeManager?token=...&authenticationContextId=...

Endpoint	HTTP method and URL	
Authorization	GET	https://accounts.google.com/o/oauth2/v2/auth?redirect_uri=...&client_id=...&response_type=code&scope=...
Client's redirection	GET	redirect_uri?code=...
Access token request	POST	https://oauth2.googleapis.com/token
Attribute request	GET	https://www.googleapis.com/oauth2/v3/userinfo

that our VPN server was able to authenticate anyone with a Google account.

III. WIREGUARD VPN SETUP

A WireGuard VPN is essentially a secure IP tunnel between two or more peers implementing WireGuard network interfaces [6]. These implement the concept of Cryptokey Routing, where each interface only needs to know its peer interfaces public keys and tunnel IPs [7].

In Linux systems, the WireGuard interfaces are configured using command line tools and configuration files. In order to connect a VPN client to a VPN server, we first need to create asymmetric key pairs for each one of them. To do so, we can run

```
wg genkey > privkey
wg pubkey < privkey > pubkey
```

on each machine, generating two pairs of keys. These are Curve25519 key pairs, which are used to run Elliptic Curve Diffie-Hellman key distribution protocols [2].

A configuration file for a WireGuard server (e.g., `wg0.conf`) would have a structure as follows:

```
[Interface]
PrivateKey = server private key
ListenPort = UDP port to listen for clients
Address = server VPN IP address / netmask bits

[Peer]
PublicKey = client public key
AllowedIPs = traffic to tunnel to/from the peer
```

This configuration essentially declares that the WireGuard's VPN server interface will have the indicated private key and tunnel IP address, and will use a given UDP port to interact with the peers.

Several peers can be indicated for each interface, and each is identified by its public key. The peers's tunnel UDP/IP addresses are not fixed, they can even vary over time (peers can roam). The tunneled traffic from peers, however, must come from the allowed IP addresses. Similarly, the interface should be used to tunnel all traffic to those IP addresses. The list of IP addresses in `AllowedIPs` is a routing table when choosing an interface for outbound traffic, and an access control list for filtering inbound traffic.

For the client, the configuration file would be:

```
[Interface]
PrivateKey = client private key
Address = client tunnel IP address / netmask

[Peer]
PublicKey = server public key
Endpoint = server initial t unnel UDP/IP port
AllowedIPs = traffic to tunnel to/from the peer
```

This configuration declares that the WireGuard VPN client interface will have the indicated private key and tunnel IP address and will communicate with a peer (server) with the provided public key and tunnel UDP/IP endpoint. It also declares, through the `AllowedIPs` parameter, which traffic should be routed to that peer (destination addresses matching the parameter) and, vice-versa, which traffic from the peer can be accepted (source addresses matching the parameter).

WireGuard can be used in scenarios where both hosts can act like client and server to each other, thus peers. In that case, they both should have a configuration similar to the server's one. However, that is not our case; we are considering a scenario where a user (client) establishes a VPN to a server. In this case, they are not peers *stricto sensu*.

In Linux, the WireGuard interfaces can be set up by storing the configuration files on the folder `/etc/wireguard` and running the command

```
wg-quick up wg0
```

This means that the set up of WireGuard VPN endpoints can be done in a simple way, just by running a few commands, upon knowing some of its peers attributes, namely their public keys and tunnel IPs and UDP ports.

In our approach, we start from a minimum of shared knowledge to initiate a VPN: the VPN server hostname and a certified public key, for the server, and a user identity attribute, for the client. Note that this information is not related with the identification elements that WireGuard uses. It is during our setup protocol that we exchange, in a trustworthy way, the public keys of both WireGuard endpoints. Once exchanged, the keys (and the IP addresses being used so far) are stored in configuration files and both sides initiate the VPN.

Copyright (c) IARIA, 2023. ISBN: 978-1-68558-052-0

IV. Proposed approach

The architectural approach that we propose to instantiate WireGuard-based VPNs with an IdP-based user authentication and identification is briefly summarized in Figure 1, and described in more detail along the rest of this section.

A. Exploitation of different IdPs

Our main goal was to create a VPN upon an OAuth 2.0 user authentication with one external IdP. Therefore, we need to adapt our proposal to the way existing IdPs deal with OAuth 2.0.

As we saw in Section II-B, the two OAuth 2.0 flows that can be used by IdPs are authorization code flow and implicit flow. And they are both used, in fact. Therefore, we looked for a solution that could work with both flows, while keeping the system flexible to evolution.

Since some IdPs are very bureaucratic for the registration of new OAuth 2.0 clients (notably Autenticação.gov), we decided that, in the worst case, we could have to have a single client registration for all our VPN server installations. Therefore, we decided to take the OAuth 2.0 for Native Apps approach [4] and use an universal IP address, a localhost address (e.g., 127.0.0.1), for the OAuth 2.0 redirection endpoint for all IdPs. This endpoint is handled by the VPN client (see Figure 1), but it is not the real OAuth 2.0 client (it is the VPN server). Consequently, the VPN client forwards all the data received in that endpoint to the VPN server.

The use of a localhost address for an OAuth 2.0 redirection endpoint requires acceptance by the IdP on the client registration. However, both the IdPs that we have used (Autenticação.gov and Google Identity) accept it. Therefore, it may not be an architectural limitation.

The VPN server is able to work with several IdPs, and presents their list for the user upon an initial VPN setup request (through the Web browser). This list can vary for different VPN servers, and grow with time, and the server's code may have to be updated for dealing with new IdPs. The VPN client, on the contrary, is completely agnostic about the IdPs used. Thus, it can deal with different evolutions of VPN servers regarding the IdPs supported by them.

The user identification by the VPN server depends on the IdP selected by the user, from a list provided by the server. Different IdPs may provide different identity attributes, therefore the VPN server needs to maintain a list of attributes to be requested per IdP, and also a list of attributes known by each IdP for each enrolled user. For example, for Autenticação.gov we used the user Portuguese Civil Identifier, whilst for Google Identity we used the user e-mail address.

B. Exploitation of WireGuard

Since we chose to explore WireGuard VPNs upon a IdP-based user authentication and identification, our architecture necessarily involves a trustworthy exchange of the WireGuard's client public key within the protocol used to authenticate the VPN server host and the VPN user.

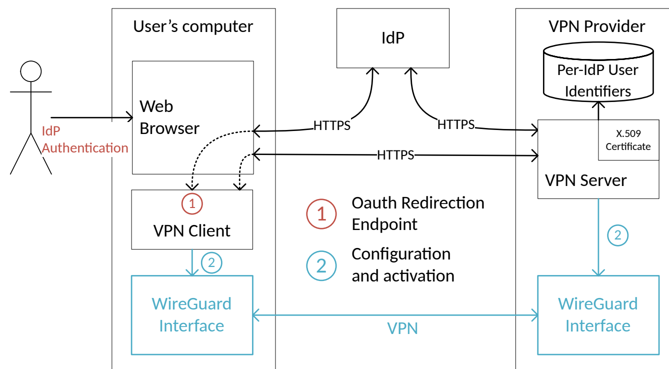

Fig. 1. High-level architecture and communication of our VPN solution

Furthermore, since our architecture requires the VPN server to handle HTTPS session, and these require a server-side X.509 public key certificate, we decided to separate the WireGuard's server side public key from the HTTPS certified public key. The first can be created on a needed basis (e.g., one per client), and are exchanged during the VPN setup, while the second, the certified public key, is expected to remain constant during the lifetime of its certificate.

Thus, our VPN client and server are applications that run a Web-based protocol that we partially designed, involving a user Web browser and an external IdP, which are able to configure WireGuard interfaces from scratch and initiate them to create a VPN. Figure 1 illustrates this approach.

C. Architecture overview

Our VPN client uses a (local) Web browser to initiate the VPN setup protocol for the current user (see Figure 2); this is required because of the way IdPs are normally explored. It also has a Web API (on localhost) to receive HTTP requests from external entities (IdPs and VPN server), redirected by the Web browser, in order to receive data required for the WireGuard setup. It does not participate on the user authentication; that is a responsibility of the IdP, and involves only the user and their Web browser.

The VPN server has a Web API accessible through HTTPS. The certificate used in the HTTPS server endpoint is the element that enables users to confirm that they are dealing with the right VPN server host. The certificate verification is performed by the users' Web browser.

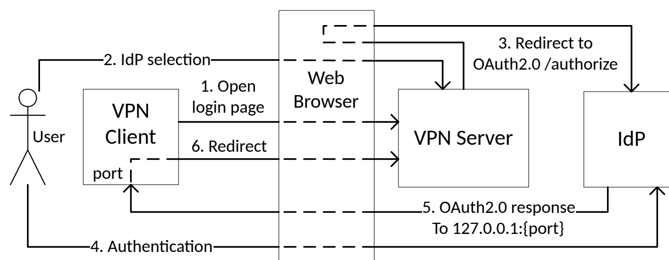

Fig. 2. Sequence of interactions for authenticating a user with an IdP

Copyright (c) IARIA, 2023. ISBN: 978-1-68558-052-0

The user identification phase, which follows the user authentication by an IdP, requires the cooperation of the VPN client and server, and also uses the user's Web browser (for HTTP redirections). However, the VPN server is the sole entity responsible for retrieving the user's identity attributes from the IdP chosen by the user. Only upon the user identification, and the necessary authorization verification, the VPN server sends the server-side WireGuard interface parameters (public key and IP address) to the VPN client, allowing it to initiate the setup the intended WireGuard client interface.

All the communication between the VPN client and server is mediated through the user browser, and every communication between the user browser and the VPN server uses HTTPS. Therefore, no sensitive information (authorization codes, access tokens, keys, etc.) is sent in clear through the Internet.

D. VPN client Web API

The VPN client handles two HTTP endpoints in a single, variable TCP port, associated to a localhost IP address (see Figure 3).

The first endpoint is the OAuth 2.0 redirection endpoint (URL with path `/login_callback`). This endpoint is the one that receives the result of the user authentication on the selected IdP (through an HTTP GET redirection). The VPN client redirects the HTTP request to a VPN server endpoint (URL with path `/login_callback`), possibly with some parameters received in the request as part of a query string. The public key of the VPN client WireGuard interface is also provided as a parameter in the redirection to the VPN server.

When the IdP uses the access code grant flow is used, the URI contains the access code in one parameter of the URI query string (`code` [10, §4.1.2]). This parameter is added to the query string used for the VPN server redirection.

When the IdP uses the implicit flow, the access token is conveyed as a URL fragment [10, §4.2] (the last part of a URL, initiated by a `#` character). Since URL fragments are retained by browsers, and not conveyed to HTTP servers, the VPN server cannot immediately get the access token from the VPN client redirection. In this case, the VPN server needs to provide the user browser with a JavaScript-enabled resource that could read the fragment contents from the URL and upload them to the VPN server.

The second endpoint is used by VPN servers to add themselves, as WireGuard peers, to the a client WireGuard interface (URL with path `/vpn_parameters`). Upon a successful user identification, the VPN server makes a GET request to this local endpoint (through an HTTP redirection) carrying in the URL query string all the parameters required to set up a peer to an existing WireGuard interface: the server public key, the server UDP/IP port and the traffic to route to the server.

E. VPN server Web API

The VPN server handles three HTTP endpoints in a single, public HTTPS port (see Figure 3).

The first is the initial login endpoint, with a URL path `/login`. The VPN client launches a browser with this endpoint to allow the user to chose an IdP to authenticate with. As a query string parameter, the VPN client provides its localhost TCP port, so it can be included in the request to the IdP as part of the redirection URI. The response includes an HTTP cookie, which contains that port. This is done in order to retrieve the port later when the user actually chooses the IdP.

Client HTTP endpoints
`login_callback[?code=...]` `vpn_parameters?pubkey=...&endpoint=...&ips=...`

Server HTTPS endpoints

`login?port=...`
IdP-specific login (e.g., `login/<IdP name>`)
`login_callback?pubkey=...[&code=...]`

Fig. 3. Web API of the VPN client and server. The parameters within square brackets are optional.

The second is the IdP-specific login endpoint. This is the endpoint which is requested when the user chooses a particular IdP. VPN servers can choose them freely, since they include them in the HTML resource presented to the user as result of the call to the generic login endpoint. The VPN server uses this endpoint, alongside with the port encapsulated in the cookie, to redirect the user browser to the chosen IdP Authorization endpoint, allowing the user to authenticate. The response also includes a new cookie with the name of the chosen IdP.

The third is login callback endpoint, with the URL path `/login_callback` (already referred in the previous section). This endpoint is where the VPN client redirects the response given by the IdP, possibly an authorization code as an URL query parameter. The cookie containing the IdP that was used will help the VPN server to select the appropriate approach to take in order get an access token from that authorization code or from the URL fragment that was kept in the user browser. Once having the access token, and knowing the IdP being used, the VPN server can request the necessary user identity attributes from the IdP, finalizing the user identification process.

V. IMPLEMENTATION

The focus regarding the implementation of the above approach was the IdP-based user authentication and identification, since the WireGuard VPN tunneling setup is trivial as long as the peers know each others public keys and IPs.

This section will start by explaining the implementation of the VPN client and server, followed by their interaction with the users, with their Web browser and with external IdPs.

A. VPN client

The VPN client is a Flask [15] application that runs in an arbitrary localhost TCP port (127.0.0.1:port). It starts by firing up a browser and opening the VPN server's login page, through HTTPS, with the query argument `?port=port`. It then listens to requests from the remaining entities in its two HTTP endpoints (`login_callback` and `vpn_parameters`).

B. VPN server

The VPN server is a Django [5] application. Its code is mostly generic for all IdPs. The handling of different IdPs happens when the user browser is redirected to the IdP selected by the user (each IdP has a specific URL for this purpose) and when the VPN server needs to get the user identity attributes from an IdP upon receiving a request on the `login_callback` endpoint.

In the last case, the code needs also to handle different OAuth 2.0 flows. When handling OAuth 2.0 implicit grant flows, in which an access token is directly provided by the IdP, the only thing to do is to build the specific request for the IdP to retrieve the user's identity attributes. When handling OAuth 2.0 access code grant flows, in which an authorization code is first provided by the IdP, these require the VPN server (OAuth 2.0 client) authentication by the IdP (using the registered credentials) to get an access token, first, and then a request with the access token to get the user's identity attributes.

However, it is not possible to provide an abstraction for the retrieval of the user's identity attributes, not even per flow type, since each IdP implements that process in their own particular way, within the OAuth 2.0 framework boundaries. Therefore, the VPN server must know each IdP particular way to implement the respective OAuth 2.0 flow. Furthermore, the user identity attributes provided by each IdP are also different.

C. Detailed communication

When the user wants to login on a VPN server, they execute the VPN client with the hostname (and possibly a port) of the server's login endpoint. The VPN client fires up the user's browser with the VPN server login endpoint plus with the VPN client's localhost port as a URL parameter. This port is returned encapsulated in a cookie of the server's response, in order to be maintained along the following browser-server HTTP interactions (cookie1 in Figure 4).

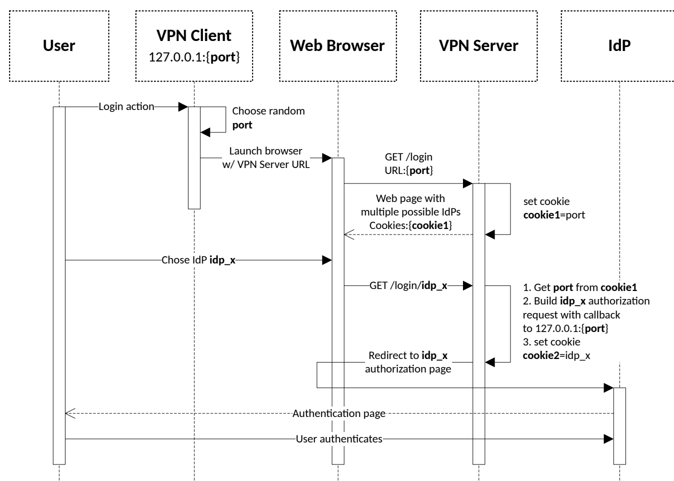

Fig. 4. VPN setup initial phase: IdP-based user authentication

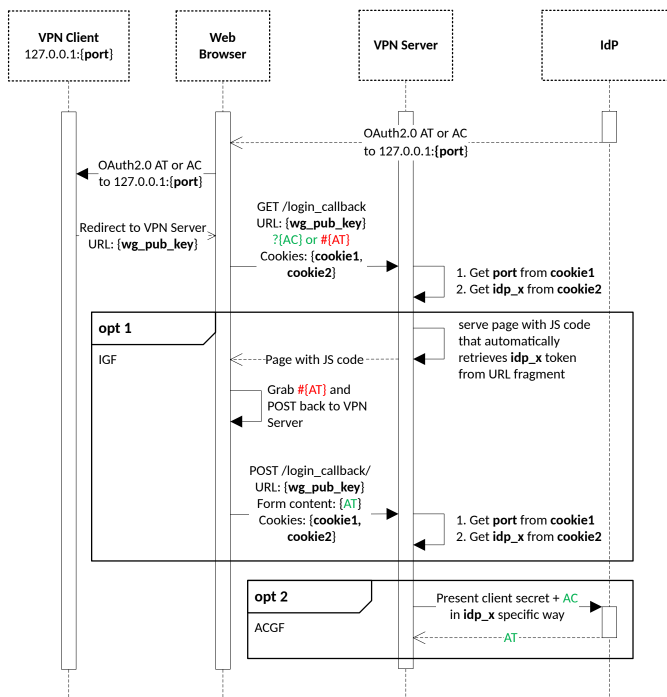

Fig. 5. OAuth 2.0 access token (AT) retrieval by the VPN server. The AC acronym stands for access token. The first optional interactions take place when the implicit grant flow (IGF) is used. The second optional interaction take place when the authorization code grant flow (ACGF) is used.

The VPN server response contains a Web page with all the possible IdPs that the user can use to authenticate with. According to the user's choice, a new request is made to the VPN server, which builds the appropriate OAuth 2.0 authorization redirection. Alongside with that redirection, another cookie is set in the user browser (**cookie2** in Figure 4), which contains the name of the chosen IdP. The user then authenticates with the chosen IdP. Figure 4 illustrates the protocol until this point.

The next phase is illustrated by Figure 5. After the user authentication, the IdP responds with an HTTP redirection to the redirection endpoint provided by the VPN server (which must conform with the registered one). This endpoint is a URL which is always uses the localhost IP address 127.0.0.1, combined with the port indicated by the VPN client. The parameters in the IdP response, together with the public key of the VPN client WireGuard interface, are then redirected to the VPN server.

If the VPN server receives a request to the `login_callback` endpoint, it expects that request to be a redirection from an IdP. The IdP is identified by **cookie2**, which makes it possible to the VPN Server to know which type of OAuth 2.0 flow that IdP implements (IGF or ACGF). If the VPN Server identifies an IdP that implements the IGF (optional block 1 in Figure 5), it returns a page containing JavaScript code that retrieves that specific access token, since URL fragments do not leave the browser. That JavaScript code will automatically run when the page is loaded, and will place the URL fragment token inside a

Copyright (c) IARIA, 2023. ISBN: 978-1-68558-052-0

Form, which will be automatically submitted back to the VPN server.

If, on the contrary, the VPN server identifies an IdP that implements ACGF, and if the request contains an authorization code, it uses it, with the appropriate HTTP client authentication, to receive an access token (optional block 2 in Figure 5).

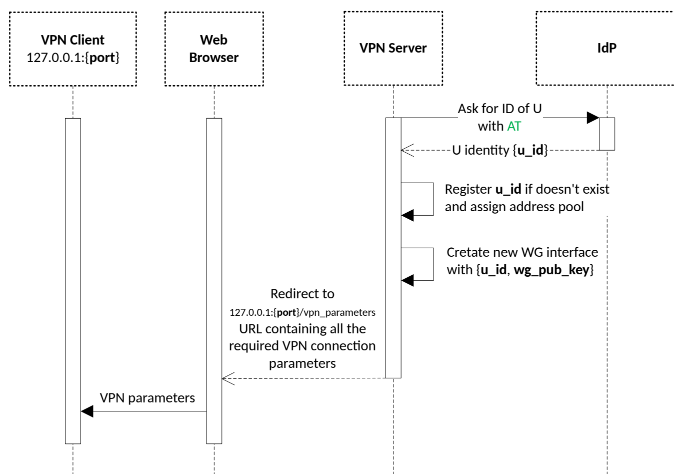

Fig. 6. User identity retrieval and configuration of the peer information in the user's WireGuard VPN endpoint

The final phase of the protocol deals with the user's identity attributes and with the setup of the WireGuard VPN, and is illustrated by Figure 6.

Once having the access token and knowing the IdP selected by the user (from **cookie2**), the VPN server retrieves the user identity attributes predefined for that IdP. When the identity attributes are received, the VPN server checks if they belong to an authorized user (registered with a set of attributes per IdP).

If the user is properly registered, the VPN server sets a peer to its WireGuard interface with the public key provided by the client and activates its WireGuard interface. Then, the VPN server sends a response with an HTTP redirection to the user's VPN Client containing the WireGuard setup information regarding its new peer (the server's WireGuard interface). This information includes the server WireGuard public key and IP. Once the configuration made, the VPN client can activate its WireGuard interface, allowing the peers to connect.

D. Testing

The solution underwent a timing analysis to evaluate its performance. The analysis focused on measuring the redirection delays between the VPN client and server, and the time to perform a login and create a WireGuard interface by the VPN server. The delays associated with the mandatory steps of the OAuth 2.0 framework and the user's authentication with the IdP were excluded as they are arbitrary and vary according to the IdP used. However, due to the lack of space in this document to present the results, and also due to the fact that the observed time figures are all less than a second, thus irrelevant in a login operation, we do not provide further details.

VI. DISCUSSION

This section is dedicated to a discussion regarding the security and usability of the proposed solution. As the main goal is to have a VPN between a user host and a server (or network gateway), at the expense of having identity attributes involved, every confidentiality and trust aspect must be taken into account. Also, the solution should be easy, simple and secure for the user to benefit, and for the VPN server provider to implement and deploy.

A. Security

Our threat model excludes attacks against to and from the IdPs, because they are assumed to be trustworthy for the service they provide.

The threat model also excludes attacks against the VPN server from its host, because, when comparing with other solutions, we mainly remove the user authentication from it, therefore we reduced its attack surface. Furthermore, the OAuth 2.0 client credentials that it holds for all the registered IdPs are not critical for itself and for the VPN users, since they only allow it to fetch attributes from an IdP for a given user upon a proper interaction between that user and the IdP (therefore, with the consent of the former).

Finally, we also exclude attacks against the VPN client from its host, because otherwise, we would have to ultimately assume that user authentication credentials could be stolen and used by attackers to impersonate them.

Therefore, we assume that the threat model includes attacks against the exchanged messages (eavesdropping, tampering, or replaying) or against the communication endpoints. And we also assume that attackers may try to impersonate legit users or legit VPN servers. Finally, we assume that malicious VPN servers may attempt to steal user-related OAuth 2.0 data items provided by browsers in order to impersonate the associated users.

Confidentiality and integrity is assured between the VPN client and the VPN server, since every communication between the two entities is done over HTTPS (HTTP over TLS, mediated by the browser). The same happens between the VPN server and the external IdP. Thus, every communication regarding the VPN setup process between these three entities is properly ciphered, its integrity is assured and the server endpoints are authenticated with X.509 certificates.

Since HTTPS security is built upon using X.509 certificates, which provide a trustworthy binding between host names and public keys, the VPN client can be sure it is dealing with the intended VPN server. Thus, both VPN client and server can trust on the WireGuard configuration elements provided to each other, namely their interfaces' public keys.

Similarly, the VPN server can verify it is interacting with the correct IdPs, selected from a list that it provides and in which it trusts.

Regarding the correctness of the authorization code grant or access token received by the VPN server, that depends on the way the client registered on each IdP. In particular, they can only be fully trusted if the VPN server has a unique client ID

Copyright (c) IARIA, 2023. ISBN: 978-1-68558-052-0

and client secret shared with the IdP. Otherwise, if there is no client secret (implicit grant flow) or if the same client secret is used by all VPN server instances, then the credentials the VPN server receives from the IdP via the user browser may have been stolen from previous, similar interactions.

The steeling of IdP-provided credentials can occur in different ways.

When the IdP uses the implicit grant flow, the VPN server trusts on the correctness of the client user solely based in the presentation of a OAuth 2.0 access token, which is a bearer token. These tokens can be stolen inside the user's machine (e.g., by another application running locally [4]) or by an malicious agent running a server for which a similar access token was fetched by the user. For instance, with Autenticação.gov, from which we require the provisioning of a user's unique identifier (Portuguese Citizen Identifier), any malicious server requiring a similar access token from the user can use it later to impersonate the user in an access to an instance of our VPN server during a period of time defined by the IdP. However, this is not a problem of our solution, this is a problem of the way the IdP works.

This problem is similar when the authorization code grant flow is used with a well-known client secret (all VPN server instances use the same). In this case, stealing the authorization grant code from a similar interaction with the IdP (one involving the same client ID) allows its reuse in another instance of the client with the same client ID. Thus, a tampered version of our VPN server deployed by a malicious agent could reuse the access code grants received from users to impersonate them in the access to other similar VPN server deployments. Alternatively, the authorization code grant can be stolen in the user host by some malicious software and reuse to impersonate the user in the access to similar VPN server instances (they need to use the same client ID, otherwise the authorization code is useless).

The usage of the Proof Key for Code Exchange (PKCE) protocol [17] mitigates this last problem, since it implements a proof-of-possession extension to OAuth 2.0 that protects the authorization code from being used when stolen. It works by adding a code challenge to the first request to the IdP. This challenge results from the hashing of a random code verifier. The IdP stores the code challenge together with the provided authorization code. The client, then, sends the code verifier when requesting the access token, and the IdP verifies if hashing it produces the code challenge. Since hashing function are one-way functions (not invertible), only a legit client, with the original code verifier, can get the access token.

Google Identity supports PKCE [8], and we used it in our prototype, although we did not describe that step in Section V-C because the previous discussion was necessary to understand its relevance. The code verifier is generated by the VPN server prior to redirecting the control to the IdP, and stored in a ciphered cookie that it sends along with cookie2 (see Figure 5). The cookie encryption uses a secret key known only by the VPN server, created each time it is launched. This cookie will later be received along with the authorization code, and the embedded code verifier can then be used to request the access token.

Wrapping up, our VPN server can have more trust in the identity of the user when the authorization code grant is used, either with a client ID and secret per VPN server instance (possible with Google Identity, for instance) or with a client ID and secret shared by many VPN server instance, provided that PKCE could be used. On the other hand, when the implicit flow is used, there is more room for user identity stealing. In that case, the user identification by an IdP could be used mainly as a second factor authentication, in order to reduce the chances of impersonation.

B. Usability

In this field, we can discuss the usability of the proposed solution for both the users of the VPN client and for the VPN server provider.

Regarding the users, this solution is an advantage since it does not require learning a new authentication interface. In this case, the users will authenticate themselves using a Web interface which they already know and trust, while other VPN solutions rely on their own custom made interfaces, which are unknown to the users and always require some learning.

The proposed solution also provides a simple and scalable deployment strategy to the VPN server provider, since the VPN server requirements regarding user's identity are delegated in the possession of OAuth 2.0 credentials from the external IdPs. Redirect URLs do not need to be previously established with the IdPs, since they are always local to the VPN client. Regarding the VPN client instances, these just need to know a *priori* the VPN server domain.

VII. Related Work

In [11], the authors propose to authenticate VPN users with a X.509 certificate containing a SAML assertion as extension. This assertion contains the identity attributes required by the VPN server to authorize the user to create the VPN [12]. This solution requires a custom Certification Authority to issue those special certificates. Those certificates must also be obtained prior to instantiate the VPN.

A solution that supports VPN authentication using IdPs is Tailscale. With Tailscale, users can create VPNs that allow them to securely access resources on remote networks, as well as share files, printers, and other resources with other users on the network [16]. It is essentially a VPN software based on WireGuard that allows user authentication with some existing OIDC-based IdPs out of the box (Google Identity, Azure AD and GitHub) and with two SAML and OIDC IdPs (Okta and OneLogin). It also allows the integration of custom OAuth 2.0, SAML or OIDC providers [18]. This integration requires manual work and configuration and is only available through their paid Enterprise subscription [19].

VIII. Conclusions

This paper describes a VPN solution that resorts to external IdPs to authenticate client users. For the low-level

VPN implementation, we chose WireGuard, which supports a manual setup of the peers. The protocol designed for user authentication also distributes the critical elements (public keys and tunneling UDP/IP ports) that should be used for creating a WireGuard VPN. The majority of the user interface is handled by a Web browser, which is the usual tool the users use when they are authenticated by an IdP.

Our VPN solution was implemented with two IdPs, Autenticação.gov and Google Identity. They both support the use of OAuth 2.0 to authenticate people and fetch some of their identity attributes. However, they explore OAuth 2.0 in different ways, which were all considered in our architecture and tackled in the implementation.

Since each IdP can explore OAuth 2.0 in a different way, the code of our solution needs to be modified. Currently, we do not have a modular approach that could be used to add new IdPs while keeping the core system stable, but that can be done.

The security of IdP-based user identification depends on the way IdPs explore OAuth 2.0. We discussed some strategies and saw that some are weaker, namely the implicit grant flow. In that case, the user authentication by an IdP should be complemented with another authentication mechanism, to implement a two-factor authentication.

As a proof of concept, we implemented the system using Flask (for the client) and Django (for the server) and we deployed a VPN for tunneling all traffic from a user laptop to a VPN server deployed in the cloud, which would later route it to the Internet.

Acknowledgment

This research work was funded by Portuguese National Funds through the FCT - Foundation for Science and Technology, in the context of the research grant BI/DETI/9308/2022 within the project UIDB/00127/2020.

References

[1] Agência para a Modernização Administrativa. *Documentação técnica relativa ao serviço de autenticação do Autenticação.Gov e Chave Móvel Digital*. 2022. URL: https://github.com/amagovpt/doc-AUTENTICACAO (visited on 01/23/2023).

[2] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *Public Key Cryptography (PKC 2006, LNCS 3958)*. Ed. by Moti Yung et al. Springer, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. DOI: 10.1007/11745853_14.

[3] Bundesministerium für Finanzen. *ID Austria: Mein Ich-organisiere-das-von-überall-Ausweis*. URL: https://www.oesterreich.gv.at/id-austria.html (visited on 01/23/2023).

[4] W. Denniss and J. Bradley. *OAuth 2.0 for Native Apps*. RFC 8252 (Proposed Standard). Oct. 2017. DOI: 10.17487/RFC8252.

[5] Django Software Foundation. *Django: The web framework for perfectionists with deadlines*. URL: https://www.djangoproject.com/ (visited on 01/23/2023).

[6] Jason A. Donenfeld. *WireGuard: fast, modern, secure VPN tunnel*. URL: https://www.wireguard.com (visited on 01/23/2023).

[7] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *Network and Distributed System Security Symposium (NDSS'17)*. San Diego, CA, USA, Feb. 2017, pp. 1–12. DOI: 10.14722/ndss.2017.23160.

[8] Google. *Using OAuth 2.0 to Access Google APIs*. URL: https://developers.google.com/identity/protocols/oauth2 (visited on 01/23/2023).

[9] Government of Spain. *Get to know Cl@ve: Electronic Identity for the Administration*. URL: https://clave.gob.es/clave_Home/en/clave.html (visited on 01/23/2023).

[10] D. Hardt (Ed.) *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Oct. 2012. DOI: 10.17487/RFC6749.

[11] Eva Hladka et al. "Transparent security for collaborative environments". In: *Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2007)*. 2007, pp. 79–84. DOI: 10.1109/COLCOM.2007.4553814.

[12] Petr Holub et al. "Secure and pervasive collaborative platform for medical applications". In: *Studies in Health Technology and Informatics* 126 (2007). PMID: 17476065, pp. 229–238.

[13] John Hughes et al. *Profiles for the OASIS Security Assertion Markup Language (SAML) 2.0*. Mar. 2005. URL: http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.

[14] OASIS (Organization for the Advancement of Structured Information Standards). *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. Committee Draft 02. Mar. 2008. URL: https://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf.

[15] Pallets Projects. *Flask web development, one drop at the time*. URL: https://flask.palletsprojects.com/en/2.2.x/ (visited on 01/23/2023).

[16] Avery Pennarun. *How Tailscale works*. Mar. 2020. URL: https://tailscale.com/blog/how-tailscale-works/ (visited on 01/23/2023).

[17] N. Sakimura (Ed.), J. Bradley, and N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. RFC 7636 (Proposed Standard). Sept. 2015. DOI: 10.17487/RFC7636.

[18] Tailscale. *Custom SSO providers using SAML or OIDC*. July 2022. URL: https://tailscale.com/kb/1119/sso-saml-oidc/ (visited on 01/23/2023).

[19] Tailscale. *Pricing*. URL: https://tailscale.com/pricing/ (visited on 01/23/2023).