Work in progress

This is an early draft of a chapter of a book on the foundations of cryptographic authentication being coauthored by Francisco Corella, Sukhi Chuhan and Veronica Wojnas. Please send comments to the authors.

5. Web technology

Chapter summary

In this chapter we describe the aspects of Web technology most relevant to cryptographic authentication. In Sections 5.1.1 and 5.1.2 we describe the basics of the technology: the HTTP protocol, the structure of URLs, the structure of Web pages, the DOM, and the web origin concept. In Section 5.1.3 we describe the two kinds of redirection, HTTP redirection and JavaScript redirection, explaining the evolution of HTTP redirection and the historical confusion resulting from that evolution. In Section 5.1.4 we describe the Web Cryptography API, pointing out its benefits and drawbacks, the web workers API and the Service Worker API, which is the basis for the browser-as-a-wallet concept, and the Web Storage and IndexedDB APIs, referring the reader to Chapter 10 for information about the Web Authentication API. Then in Section 5.2 we explain the concept of using delegated authorization for federated identity management, which provides the founcation for the OAuth2 protocol, described in Section 5.3, and for OpenID Connect, described in Section 5.4. Finally, in Section 5.5 we describe the JSON Web Tokens used in OpendID connect and related protocols.

5.1 The World Wide Web

5.1.1 HTTP and URLs

The web allows users equipped with web browsers, formally known as user agents, to access web pages and retrieve resources such as documents and media from web servers over the internet using the application layer Hypertext Transfer Protocol (HTTP) [1] over a connection, which may be TCP, but these days it is more likely to be TLS over TCP.

A web browser communicates with a web server, or a web server with another web server, by sending an *HTTP request* addressed to a Uniform Resource Locator (URL) [2] over a connection and receiving an *HTTP response* over the same connection.

A URL consists of a *scheme*, which may be *http* for a TCP connection or *https* for a TLS connection, followed by the *hostname* of the server, which is usually the DNS domain associated with the IP address used by the physical server, optionally followed by an optional *path* consisting of *segments* separated by forward slashes that may correspond to a path in the filesystem of the server, optionally followed by a question mark and a *query string* consisting of a sequence of *key=value* pairs separated by ampersands (&), optionally followed by an octothorpe (#) and a *fragment*, which refers to a position within a web page addressed by the URL, as we shall see in the next section. A URL may be abbreviated by omitting the scheme and the domain name. An abbreviated URL is called a *relative URL*, and a non-abbreviated URL is called an *absolute URL.*

An HTTP Request comprises a *request line*, *request headers*, and an optional *request body*. The request line specifies an *HTTP method*, usually GET or POST, the URL of the server, and the HTTP version used to construct the request. As of this writing, the current version of HTTP is HTTP/1.1. The URL in the request line is a relative URL, except if the request is sent to a proxy server that will forward it to its final destination, in which case the URL is the absolute URL of the final destination. The request line is followed by a sequence of headers, each on a separate line, of the form *header-name*: *header-value.* If the HTTP mode is POST, the sequence of headers is followed by a blank line and the HTTP body; GET requests do not have an HTTP body. The sequence of headers must include a header with name *Host* whose value is the hostname of the destination server. The Host header was made mandatory by version 1.1 of HTTP to allow multiple virtual servers to be hosted on the same physical server. Other commonly used headers include *User-Agent*, *Referer, Cookie* and, in a POST request, *Content-Type* and *Content-Length*, which refer to the content of the body of the request.

An HTTP response similarly comprises a *status line*, a sequence of *response headers*, and an optional *response body.* The status line comprises the HTTP version, a numeric, 3-digit, status code, and a status message. An example of a status line is "HTTP/1.1 200 OK", and another example is "HTTP/1.1 404 Not found". Commonly used response headers include *Content-Type*, *Content-Length*, and *Set-Cookie.* We saw a detailed example of the use of the Set-Cookie response header and the Cookie request header in the Section 4.3 of Chapter 4, in connection with the Evilginx attack, where the attacker impersonates the user by capturing a login session cookie.

5.1.2 Web pages

When a user visits a web page by clicking on a link to a URL, scans a QR code containing a URL, or types a URL on the address bar of the browser, the user's browser establishes a TCP or TLS connection as specified by the scheme component of the URL to the server identified by the hostname component of the URL, and sends an HTTP GET request over the connection, with a request line comprising the URL. If all goes well, the server sends an HTTP response over the same connection containing the HTML source code of the web page.

HTML markup

HTML is a markup language where marked up text is enclosed between an opening tag such as <p>, where the name of the tag is enclosed by angle brackets, and a corresponding closing tag such as </p> where the name of the tag is preceded by a forward slash and enclosed in angle brackets.

An opening tag, may have one or more attributes. For example, the tag <a>, called the anchor tag, may have an href attribute or a name attribute. When it has an href attribute, as in <a href="https://example.com/test-page.html">, the text between that opening tag and the closing tag </a> is rendered by the browser as a hyperlink. When the user clicks on that link, the browser sends an HTTP GET request to the URL https://example.com/test-page.hml, and if the test page exists, it receives an HTTP response with status line "HTTP/1.1 200 OK" and a body containing the HTML source code of the test page.

Fragments

Continuing the example, the test page may have a section with title "Test section", and that title may be enclosed between an opening tag <a name="test-section"> and a closing tag </a>. The browser does not render the title as a hyperlink, because the name attribute does not specify a target URL. Instead, it specifies a location within the page. If the link to the page has a fragment component "#test-section", as in <a href="https://example.com/test-page.html#test-section">, the browser scrolls to the test section after rendering the test page.

Since the fragment component is used by the browser after receiving the page, it is not sent to the server that hosts the page. When the user clicks on the link <a href="https://example.com/test-page.html#test-section">, the browser sends an HTTP request to the URL https://example.com/test-page.html, and the server at example.com does not see the fragment. After the browser receives and renders the test page, it searches it for an anchor tag with name attribute "test-section", and scrolls to the position of that tag. This behavior is used to implement the implicit flow of OAuth, as we shall see below in Section 5.2.

Page structure

The source code of a web page is enclosed between <html> and </html> tags and comprises a head section enclosed by <head> and </head> tags and a body section enclosed by <body> and </body> tags. The head section contains metadata such as the title of the page, shown when hovering of a browser tab, the character set used in the page, such as ASCII or UTF-8, and references to CSS style sheets. The body section includes content that is rendered by the browser and displayed in the browser page or tab, such as marked up text and tags with URLs from which the browser retrieves images and other media.

The body section may also include forms [3]. A form is enclosed in <form> and </form> tags and contains input elements that the browser renders as UI elements where the user can enter data. For example, a login page may include a login form with a username input of type "text" that is rendered as box where the user enters a username, a password input of type "password" rendered as box that hides the password as it is being entered, and an input of type "submit" rendered as a button that is used to submit the form.

Forms

A form tag has an action attribute whose value is the URL where the form data is to be sent, and a method attribute whose value may be POST or GET. The form data is a collection of name-value pairs, where the names are the name attributes of input elements, and the values are entered into the form by the user. If the method is POST, the form data is sent in the body of a POST request. If the method is GET, it is sent in a query string appended to action attribute.

JavaScript embedded in the page

The HTML code of a web page may also include or reference JavaScript code. JavaScript may be enclosed between <script> and </script> tags that are usually located in the head section but may also be included in the body section. That JavaScript code is executed when the browser finds it while parsing and rendering the html code. JavaScript code may also be referenced by the src attribute of a <script> tag. When the browser encounters the tag, it retrieves the code using a GET request and executes it. JavaScript functions known as event handlers may also be included as the values of certain attributes and be executed when certain events occur. For example, a <form> tag may have an onSubmit attribute, whose value is a function that is executed when the user submits the form. That function may check that all required inputs have been filled in and cancel submission by returning false if that is not the case.

JavaScript code can send GET and POST requests using the Fetch API [4] without the user clicking on a link or filling out a form.

Document Object Model

The Document Object Model (DOM) [5] is a hierarchical representation of a web page as a tree, where each tag is represented by a node. JavaScript code running in the page can use the DOM to inspect and modify the contents of the page.

The Web Origin concept

The web origin of a web page is, in first approximation, the DNS domain from which the page has been downloaded. More precisely, it comprises the first three components of the URL of the page: scheme, hostname, and port. The concept is "typically" used by the browser to "isolate content retrieved from different origins to prevent malicious web site operators from interfering with the operation of benign web sites" [6]. We shall see below in Section 5.1.4 how Web APIs make use of the concept.

5.1.3 Redirection

A redirection response is an HTTP response to an original HTTP request that causes the browser to immediately send a redirected request addressed to a redirection target URL different from the target URL of the original request.

There are two kinds of redirection:

- In an *HTTP redirection*, the redirection response has a redirect status code and a location header that specifies the redirection target URL. Redirect status codes begin with the digit "3".
- In a *JavaScript redirection*, the redirection response is a code-only web page with JavaScript code that sends the redirected request to the redirection target URL.

HTTP redirection has historically been confusing, as described in [7, §15.4], because it is used for two different purposes in two different kinds of use cases:

1. One purpose is the web equivalent of forwarding a letter to a new address; a typical use case where it is used for this purpose is website reorganization.

2. Another purpose is to implement multiparty protocols, where parties communicate with each other through the user's browser.

Status codes 307 and 308 support the first purpose by requiring the browser to use the method, either GET or POST in the original request and the redirected request. The method is POST, the browser must save the body of the original request and use the same body in the redirected request. Status code 303 supports the second purpose by requiring the browser to redirect a POST request as a GET request. Status codes 301 and 302 are ambiguous: some browsers will redirect a POST request as a GET request, others as a POST request with the same body. There is no ambiguity if the original request is a GET request.

JavaScript is only used for the second purpose and provides the important benefit that the redirected request can be a POST request whether or not the original request if a POST request. Sending data to the redirection target URL in the body of a POST request is preferable to sending it in the query string of a GET request because the length of the query string is constrained by the 2048 character limit on the size of a URL, and because the query string is recorded in the server log, where it may be exposed to attack.

5.1.4 Web APIs

Web APIs are application programming interfaces specified by the W3C, implemented by browsers, and usable by JavaScript code running in web pages. A list of all the available web APIs can be found in [8]. In this section we reference and briefly describe a few that are particularly relevant to cryptographic authentication.

Web cryptography API

The Web cryptography API, published as a W3C recommendation in 2017 [9], comprises:

1. Interface definitions of cryptographic functions that can be invoked by JavaScript code as methods of the *SubtleCrypto* interface, including *encrypt, decrypt, sign, verify, digest, generateKey, deriveKey, deriveBits, importKey, exportKey, wrapKey, unwrapKey.* The SubtleCrypto interface is implemented by the object *crypto.subtle*, where *crypto* is an object available in the global scope of JavaScript code running in web pages, and in the global scope of worker scripts, including service workers.
2. Descriptions of symmetric and asymmetric algorithms that can be used as parameters of the above functions. Most of the functions provide symmetric or asymmetric functionality depending on the algorithm that they are used with. Descriptions written in Web IDL [10] are provided for the following algorithms: *RSASSA-PKCS1-v1_5, RSA-OAEP, ECDSA, ECDH, AES-CTR, AES-CBC, AES-GCM, AES-KW,*

HMAC, the algorithms of the two SHA families, including *SHA-1, SHA-256, SHA-384* and *SHA-512, HKDF*, and *PBKDF2*.

3. An interface definition of a function getRandomValues() for generating cryptographically random values. The function can be invoked as a method of the *crypto* object by JavaScript code running in a web page and by worker scripts. Two notes at the beginning of [9, §10.1] provide explanations of how browsers supporting the API should implement this function. The second note clarifies that getRandomValues is not supposed to be a true random number generator that blocks when not enough entropy is available. The first note suggests using /dev/urandom as an entropy source for the pseudo-random number generator; /dev/urandom is an OS-provided special file originally introduced by Linux, but now widely available across OSes, including in Windows and MacOS.

A recent update to the API, in the form of an Editor's Draft dated 09 January 2025 [11] adds descriptions of two algorithms: Ed25519 for signature and verification using elliptic curve Curve25519 as specified in RFC 8032 [12], and X25519 for key exchange, also using curve Ed25519, as specified in RFC 7748 [13]. There had been no prior updates prior to this one since the publication of the W3C Recommendation in 2017.

Benefits and drawbacks of the Web Cryptography API

The Web Cryptography API provides two important benefits: it can take advantage of a source of entropy provided by OS, such as /dev/urandom, and it can store private keys in protected storage available to the browser but not to JavaScript code running in the browser.

But it also has two disadvantages when compared to an open source cryptographic library not built into the browser.

The first disadvantage is that browsers that implement the API do not necessarily provide all the algorithms described in the specification, since Section 18.1 of the specification states:

> "In addition to providing a common interface to perform cryptographic operations, by way of the SubtleCrypto interface, this specification also provides descriptions for a variety of algorithms that authors may wish to use and that User Agents may choose to implement."

Therefore, some users of a web application whose JavaScript frontend relies on the API may encounter errors if they happen to use a browser, or a particular version of a browser, that does not implement an algorithm used by the frontend. By contrast, a web application that uses an open source cryptographic library will include in its frontend the algorithms that it needs among those provided by the library and test its code with those algorithms.

The second disadvantage is that it is cumbersome to use, because it is an asynchronous interface based on promises. It does not make sense to use an asynchronous interface for very fast operations. It has been reported in [14] that a Macbook Book with an ARM processor can perform 32,866 ECDSA signatures per second using the NIST P-256 curve. That means it takes 30.4 microseconds to perform a signature. The reported benchmark is for OpenSSL, which is implemented in C, while JavaScript is an interpreted language, but it is hard to imagine a use case where JavaScript running in a browser would have to perform thousands of operations per second. And if such a use case is ever found, there is WebAssembly.

It would make sense for the frontend of a web application to use an open source JavaScript cryptographic library instead of the Web Cryptography API, while using crypto.getRandomValues to generate random bits, thus taking advantage of the /dev/urandom noise source, and relying on alternative ways of protecting a private key other than entrusting it to the browser. This topic is further discussed in Chapter 7.

Web workers and service workers

The JavaScript code embedded in a web page runs in a single thread of execution. This means that, if JavaScript code in a <script> tag, or in an attribute of a tag, performs a lengthy operation, the page becomes unresponsive to user interaction until the operation ends. Worker threads are separate threads of execution that can perform long operations without blocking the main thread.

There are two distinct Web APIs that can be used to create worker threads, called the *Web Workers API* [15] and the *Service Worker API* [16].

The Web Workers API is used to create worker threads for a variety of purposes. A web worker thread is created by writing a file with the JavaScript code to be executed and calling the Worker constructor, passing the URL of the file as an argument.

The Service Worker API is used to create worker threads that will intercept HTTP requests and respond to them locally. A service worker is created by writing a file with the JavaScript code that will later be used to handle intercepted requests, and calling a function register(), which is a method of a ServiceWorkerContainer interface. The register() function takes as input the URL of the file and options including the scope of the requests to be intercepted, which must be within the web origin of the script that calls the register() function. When an HTTP request is addressed to a URL within the scope, the thread request is created and may handle the request by creating a web page on the fly and passing it to the rendering engine of the browser.

Web Storage API

The Web Storage API [17] allows JavaScript code embedded in a web page to store name-value pairs as properties of two global JavaScript objects called localStorage and sessionStorage. Name-value pairs stored in localStorage by a web page can be retrieved by a web page of same origin visited in a different tab of the browser, and remain available after the browser is closed and reopened. Name-value pairs stored in sessionStorage are only available in the same tab where they have been created and go away then browser is closed.

The localStorage and sessionStorage objects are only available to JavaScript code running in the main execution context. They are not available to web workers or service workers. However, after a service worker intercepts a request and creates a web page that is rendered by the browser, JavaScript in that web page can access localStorage. We shall see in Section 12.4 of Chapter 4 how this is used for credential presentation when the browser is used as a credential wallet.

IndexedDB API

The IndexedDB API provides the frontend of a web application with a facility for storing and retrieving large amounts of data. To avoid blocking the main execution thread while manipulating large amounts of data, the IndexedDB API can be accessed by worker threads but cannot be accessed by the main execution thread. Both web workers and service workers can use the API to create object-oriented databases equipped with indexes for efficient search. The storage and retrieval operations are performed asynchronously as specified by an interface based on promises.

Web Authentication API

The Web Authentication API [18] is part of the FIDO2 standard [19], which is described in Chapter 10.

5.2 Delegated authorization for federated identity management

Facebook Connect, announced in 2008 [20], was a form of federated identity management that allowed Facebook users to “log in with Facebook” to third party websites by authorizing those websites to access their Facebook account, where they found identity information.

This was recognized as an important innovation, and an effort was started at the IETF to make it generally available rather than proprietary. The effort eventually resulted in the OAuth 2.0 Authentication Framework [21], developed by the OAuth working group, which is now working on OAuth 2.1, and the OpenID Connect specification [22], developed by the OpenID foundation. Oauth 2.0 and OpenID Connect are described below in Sections 5.3 and 5.4 respectively.

Facebook was invited to participate in the effort, and there was hope that they would adopt the resulting non-proprietary specifications. They did not, but OAuth 2.0 and OpenID Connect have been successful, nevertheless. It is not clear to what extent Facebook Connect is still being used [23].

5.3 OAuth 2.0

OAuth 2.0 [21] is a protocol that allows the user of a web service to delegate authorization to access the user’s account at the service to a third-party client, so that the client can use resources owned by the user and protected from unauthorized access by the service on behalf of the user. In an example provided by the specification, the service that protects the resources is a photo storage service, the client is a photo printing service, and the user delegates authorization to access the photo storage service to the client so that the client can print the user’s photos.

The OAuth 2.0 protocol has four flows, known as grant types. In the *authorization-code flow*, described in Section 5.3.1, the user provides consent for an *authorization server* to issue an *authorization code* to the client, which the client uses to obtain an *access token*, which it then uses to access protected resources from a *resource server.* In the *implicit flow*, described in Section 5.3.2, the authorization server provides the access token itself to the client, skipping the intermediate step of issuing an authorization code to be exchanged for the access token. In the *resource owner password credentials* grant, the client uses the user’s credentials to obtain an access token that provides limited access to the user’s account. In the *client credentials* flow, the client uses its own credential to obtain the access token.

5.3.1 Authorization-code flow

Figures 1a-1g provides an example of a run of the authorization-code flow, without mentioning every parameter of every step of the flow. The reader is referred to RFC 6749 for a more precise description.

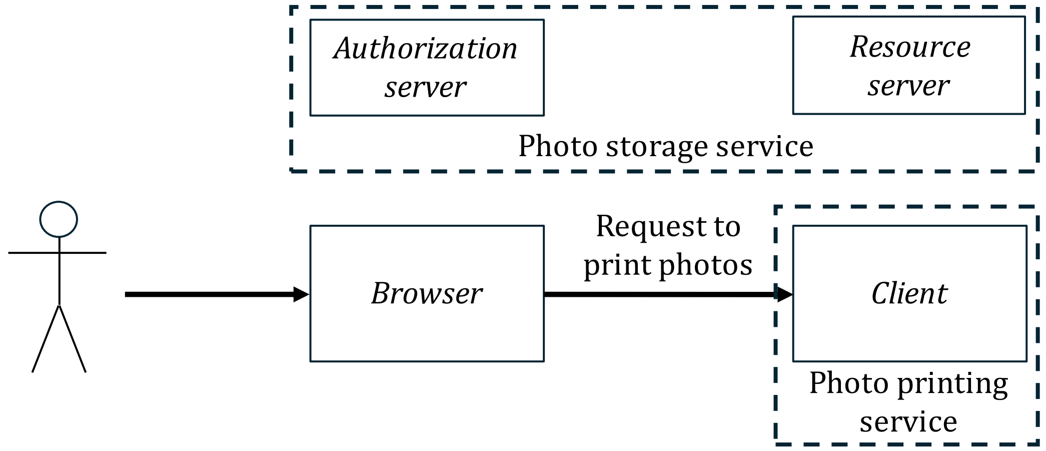

Figure 1a: User asks client to print photos stored in the photo storage service

In Figure 1a, the user submits a request to a photo printing service to print photos stored in a photo storage service. This causes the browser to send an HTTP request to the client. In this example all HTTP requests will be sent over TLS.

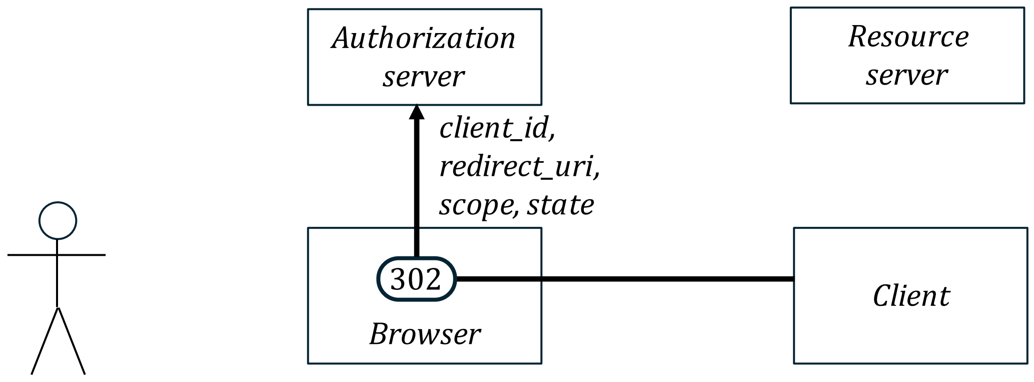

Figure 1b: Client sends an HTTP 302 redirection response

In Figure 1b, the client responds to the HTTP request of Figure 1a with an HTTP 302 redirection respond targeting the authorization of the photo sharing service, including as parameters of the query string of the location header, an ID of the client known to the authorization server, a callback URL that will be used to send the redirect response of Figure 1d, the scope of access to protected resources being requested by the client, and a client state parameter that will provide context for the redirection response of Figure 1d.

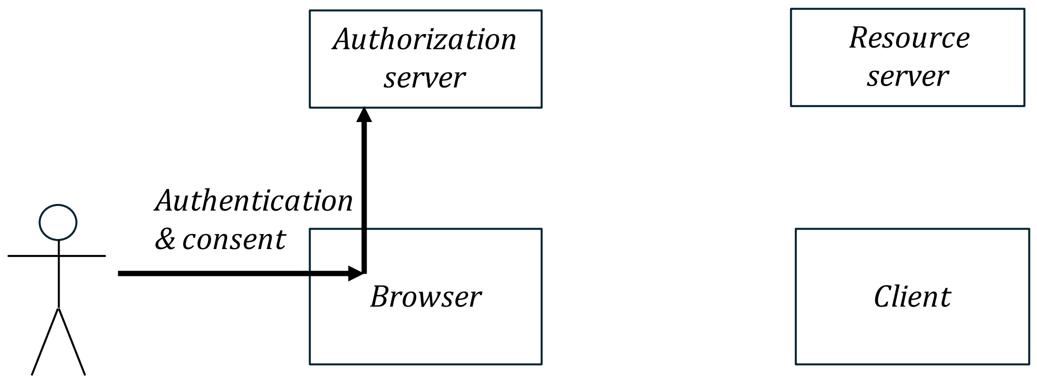

Figure 1c: User authenticates and consents to requested scope of access

In Figure 1c, the user sends an HTTP POST request to the authorization server providing authentication and consent to grant the client the requested scope of access.

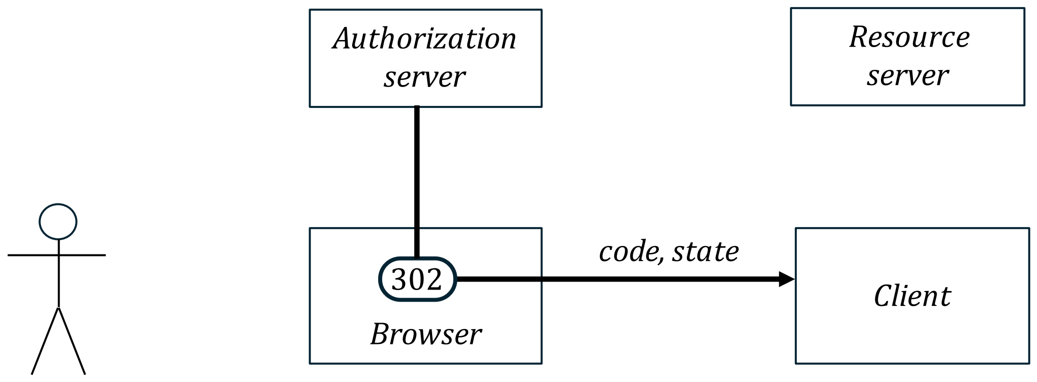

Figure 1d: The authorization server sends HTTP 302 redirection response to the client

In Figure 1d, the authorization server sends an HTTP 302 response to the HTTP request of Figure 1b, conveying the authorization code and the state that it received in Figure 1b to the client. The authorization code specifies the scope of access that was requested in Figure 1b.

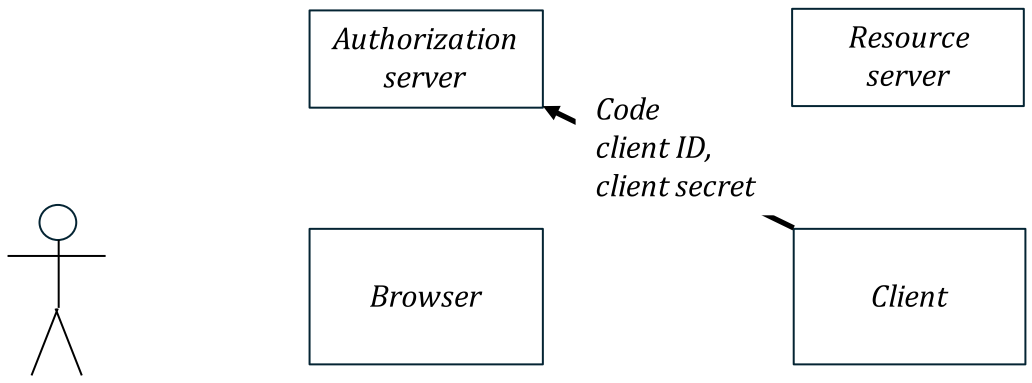

Figure 1e: Client authenticates to authorization server and submits authorization code

In Figure 1e, the client authenticates to the authorization server with its client ID and associated client secret and submits the authorization code.

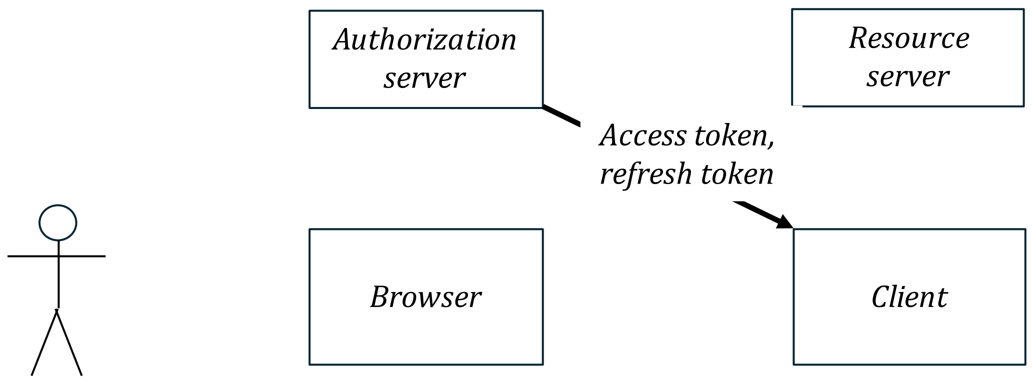

Figure 1f: Authorization server provides access token and may provide refresh token

In Figure 1f, the authorization server provides an access token for the scope of access specified in the authorization code. It may also provide a refresh token that can be used to obtain additional access tokens.

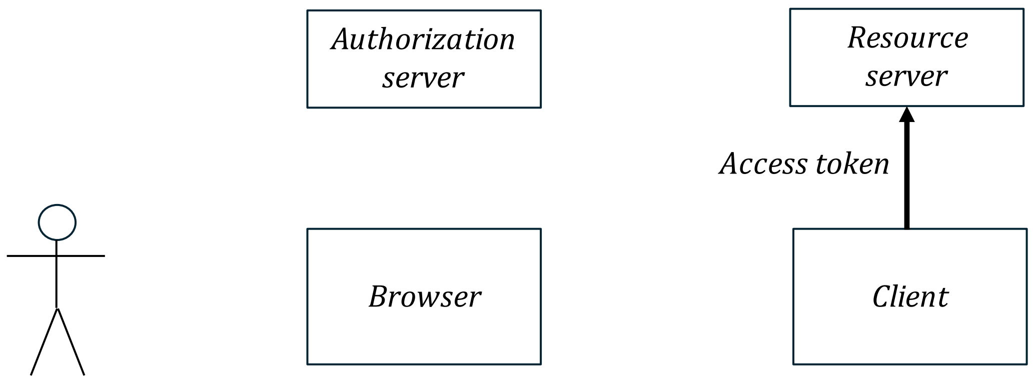

Fig1g: The client submits the accesss token to the resource server

In Figure 1g, the client submits the access token to the resource server of the photo storage service and is granted the scope of access that it requested in Figure 1b. The client can now let the user browse the photos stored in the storage services and select photos to be printed, then print the selected photos.

5.3.2 Implicit flow

In the implicit flow, the authorization server redirects back to the client with a URL that includes the access token in the fragment instead of the authorization code in query string. The steps of the flow are derived from the steps of the authorization code flow by replacing Figures 1d, 1e and 1f with figures 2a and 2b below.

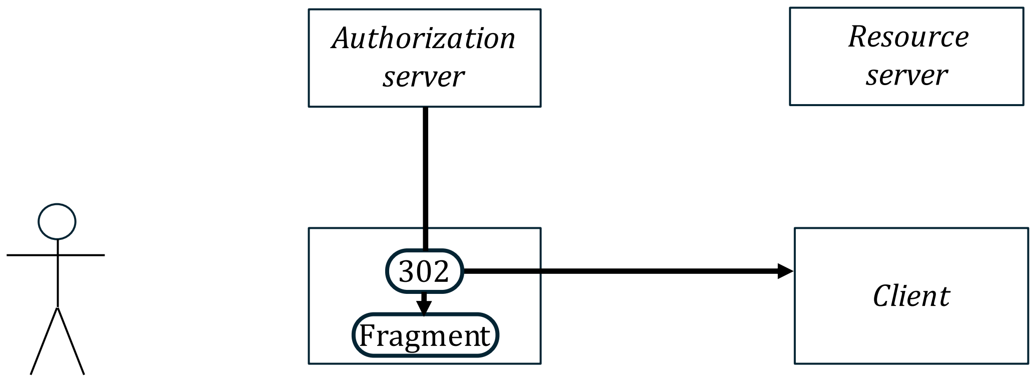

Figure 2a: The authorization server sends HTTP 302 redirection with fragment

In Figure 2a, the authorization server responds to the HTTP request of Figure 1b with an HTTP 302 response addressed to a URL with an access token included in the fragment. As we saw above in Section 5.1.1, when an HTTP request is addressed to a URL with a fragment, the fragment stays in the browser, where it is available to JavaScript running in the HTTP response to the request using the DOM, as window.location.hash, or document.location.hash, or just location.hash.

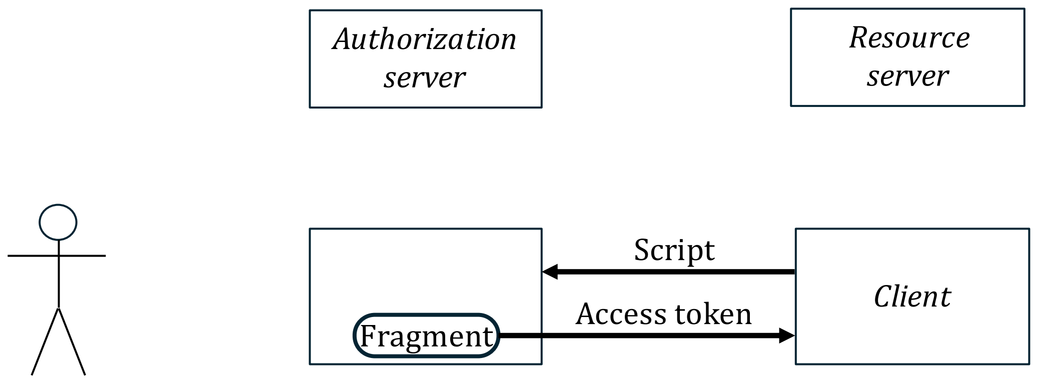

Figure 2b: The client downloads a script that retrieves the access token from the fragment

In Figure 2b, the client responds to the redirected request of Figure 2a with a code-only page containing a script that retrieves the fragment and sends it to the backend of the client in an HTTP POST request. The client parses the fragment, obtains the access token, and submits it to the resource server in the next step of the protocol, shown in Figure 1g.

The implicit flow is useful for clients that do not have a client ID and a client secret because they run in the browser, where the secret might be exposed to capture. However, the implicit flow is deemed insecure for a variety of reasons. Here is for example, what Microsoft says about the implicit flow:

> *Microsoft recommends you do not use the implicit grant flow. In most scenarios, more secure alternatives are available and recommended. Certain configurations of this flow require a very high degree of trust in the application and carries risks that are not present in other flows. You should only use this flow when other more secure flows aren't viable.*

A discussion of OAuth 2.0 security can be found in the recent RFC 9700 [24].

5.4 OpenID Connect

OpenID Connect is a successor of OpenID protocol, which was succeeded by OpenID 1.1 and then OpenID 2.0. Unfortunately the specifications of these protocols are not available on the web site of the OpenID Foundation as of this writing, with request giving 503 results.

The name of the OpenID Connect protocol is a clear reference to Facebook Connect, and as Facebook Connect it combines federated identity and delegated authentication. But it combines them differently.

Two paradigms for combining identification with authorization

Facebook used the paradigm sketched above in Section 5.2. It leveraged delegated authorization to provide identification by giving access to the user's profile to the relying party. OpenID Connect leverages instead the two subsequent redirections, from the relying party to the authorization server, then back to the relying party, to provide identification in addition to, but separately from, delegated authentication. In OpenID Connect the relying party obtains two tokens: the same access token as in OAuth 2.0, plus an ID token. The access token provides access to the user's account at the service provider until it expires, and it can be refreshed when it expires. The ID token provides a snapshop of identity information about the user at the time when it is obtained.

Protocol flows

As the specification says [22, §1], "OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol". It just returns the ID token in addition to the access token.

There are three flows:

- The *authorization code flow* is like the authorization code flow of OAuth 2.0, except that the relying party obtains the ID token in addition to the access token when it presents the authorization code to the authorization server.
- The *implicit flow* is like the implicit flow of OAuth 2.0, except that the fragment contains the ID token instead of or in addition to the access token. The relying party obtains the fragment as shown above in Figure 2b.
- The *hybrid flow* is like the authorization code flow of OAuth 2.0, and like the above authorization code flow of OpenID Connect, except that the relying party may only get one of the two tokens in exchange for the authorization code.

Signed response

The ID token is formatted as a JSON Web Token (JWT). It must be signed, and may also be encrypted, by the OpenID Provider.

It is important to notice, however, that signing it does not provide increased security because the ID token is received over TLS from the OpenID Provider. More precisely,

- In the authorization code flow and the hybrid code flow, it is received in the HTTP response to the HTTP request that the relying party sent to the authorization server at step 6 of either flow.
- In the implicit flow, it is received in the HTTP response to the HTTP request that the relying party sends to the authorization server at step 4 of the protocol.

In both cases, the mentioned HTTP request is sent over TLS with authentication of the authorization server by its TLS server certificate.

5.5 JSON Web Tokens (JWTs)

JSON [25] is a data interchange format that uses the same syntax that is used in the literal object notation of the JavaScript programming language.

JSON Web Token (JWT) [26] is a format for encoding claims as name-values pairs using JSON, where each name is a three-letter lower case abbreviation. The following claim names were registered with IANA when RFC 7519 was published:

- "iss" (Issuer)
- “sub” (Subject)
- “aud” (Audience)
- “exp” (Expiration time)
- “nbf” (Not before)
- “iat” (Issued at)
- “jti” (JWT ID)

JWTs can be signed using JSON Web Signature (JWS) [27] and/or encrypted using JSON Web Encryption (JWE) [28].

References

[1] Mozilla. HTTP. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP

[2] T. Berners-Lee et al. (1994, December). RFC 1738: Uniform Resource Locators (URL). Retrieved from https://datatracker.ietf.org/doc/html/rfc1738

[3] Mozilla. The Form element. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form.

[4] Mozilla. Fetch API. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[5] Mozilla. The Document Object Model. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[6] Barth, A. (2020, July 29). RFC 6454: The Web Origin Concept. Retrieved from https://datatracker.ietf.org/doc/rfc6454/.

[7] R. Fielding et al. (2022, June). RFC 9110: HTTP Semantics. Retrieved from https://datatracker.ietf.org/doc/html/rfc9110.

[8] Mozilla. Web APIs. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API.

[9] W3C. (2017, January 26). Web Cryptography API. Retrieved from https://www.w3.org/TR/WebCryptoAPI/.

[10] WHATWG. Web IDL. Retrieved from https://webidl.spec.whatwg.org/.

[11] Huigens, D. (2025, January 9). Web Cryptography API, W3C Editor's Draft. Retrieved from https://w3c.github.io/webcrypto/.

[12] Josefsson, S., & Liusvaara, I. (2017, January). RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA). Retrieved from https://datatracker.ietf.org/doc/html/rfc8032.

[13] A. Langley et al. (2016, January). Elliptic Curves for Security. Retrieved from https://datatracker.ietf.org/doc/html/rfc8032.

[14] Asecuritysite.com. Digital Signature Benchmark. Retrieved from https://asecuritysite.com/openssl/openssl3_b2.

[15] Mozilla. Web Workers API. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[16] Mozilla. Service Worker API. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.

[17] Mozilla. Web Storage API. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API.

[18] W3C. Web Authentication: An API for accessing Public Key Credentials. Retrieved from https://www.w3.org/TR/webauthn-2/.

[19] FIDO Alliance. FIDO Authentication. Retrieved from https://fidoalliance.org/fido2/.

[20] Morin, D. (2008, May 9). Announcing Facebook Connect. Retrieved from https://developers.facebook.com/blog/post/2008/05/09/announcing-facebook-connect/.

[21] Hardt, D. (2012, October). RFC 6749: The OAuth 2.0 Authorization Framework. Retrieved from https://datatracker.ietf.org/doc/html/rfc6749.

[22] N. Sakimura et al. (2023, December 15). OpenID Connect Core 1.0 incorporating errata set 2. Retrieved from https://openid.net/specs/openid-connect-core-1_0.html.

[23] Microsoft. Facebook Connect is no longer available. Retrieved from https://support.microsoft.com/en-us/office/facebook-connect-is-no-longer-available-f31c8107-7b5a-4e3d-8a22-e506dacb6db6.

[24] T. Lodderstedt et al. (2025, January). RFC 9700: Best Current Practice for OAuth 2.0 Security. Retrieved from https://datatracker.ietf.org/doc/rfc9700/.

[25] T. Bray. (2014, March). RFC 7159: The JavaScript Object Notation (JSON) Data Interchange Format. Retrieved from https://datatracker.ietf.org/doc/html/rfc7159.

[26] M. Jones et al. (2015, May). JSON Web Token (JWT). Retrieved from https://datatracker.ietf.org/doc/html/rfc7519.

[27] Jones, M., Bradley, J., and N. Sakimura, RFC 7515: JSON Web Signature (JWS), May 2015. Retrieved from http://www.rfc-editor.org/info/rfc7515.

[28] Jones, M. and J. Hildebrand, RFC 7516: JSON Web Encryption (JWE), May 2015. Retrieved from http://www.rfc-editor.org/info/rfc7516.