Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Proposal for Internet-Draft "FIDO2 TLS 1.3 Extension"

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von: David Wedekind
geboren am: [redacted]
geboren in:

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Ernst-Günter Giessmann

eingereicht am: ..... verteidigt am: .....

Abstract

FIDO2 offers a great alternative to passwords by relying on strong, phishing-resistant credentials for client authentication instead. As these authenticators were originally designed for the web, they do not see any use outside HTTPS applications. This thesis builds upon prior groundwork, which proposes a FIDO2 extension for TLS 1.3 as a solution to disconnect FIDO2 from its dependency on the web environment, making it available to non-HTTPS applications. The extension enables FIDO2 registration and authentication directly in the TLS handshake. To push forward a potential standardization of the extension, this thesis proposes an Internet-Draft "FIDO2 TLS 1.3 Extension", which aims to offer a precise and unambiguous definition of the extension.

Contents

Acronyms

AES Advanced Encryption Standard.

API Application Programming Interface.

ASN.1 Abstract Syntax Notation One.

BCP Best Current Practice.

CBOR Concise Binary Object Representation.

CH ClientHello.

CR CertificateRequest.

CT Certificate.

CTAP Client to Authenticator Protocol.

DoS denial-of-service.

EAP Extensible Authentication Protocol.

FIDO Fast IDentity Online.

GCM Galois/Counter Mode.

I-D Internet-Draft.

IAB Internet Architecture Board.

IANA Internet Assigned Numbers Authority.

IETF Internet Engineering Task Force.

IRTF Internet Research Task Force.

ISE Independent Submissions Editor.

JSON JavaScript Object Notation.

MITM man-in-the-middle.

POC Proof of Concept.

RFC Request for Comments.

RP Relying Party.

SAN Subject Alternative Name.

SNI Server Name Indication.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

W3C Word Wide Web Consortium.

WebAuthn Web Authentication.

1. Introduction

Pure password authentication has many problems, especially many attack vectors like social engineering, phishing and credential reuse. A promising alternative are FIDO authenticators. FIDO authenticators can be used for strong authentication without the usage of passwords by using either external hardware tokens, or so-called platform authenticators being directly integrated into the client platform.[1] As FIDO2 was originally only designed for web applications, it cannot easily be used in protocols different to HTTPS. In his master's thesis "FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks", Jonas Panizza proposes a TLS extension for integrating FIDO registration and authentication directly into TLS, making it available to non-HTTP applications. He uses OpenSSL and C to implement the extension in a working Proof of Concept (POC).[2] In this master's thesis, the goal is to disconnect the problem of a FIDO2 TLS 1.3 Extension from specific libraries like OpenSSL by writing an Internet-Draft (I-D) to try to advance a potential standardization of FIDO2 as a TLS extension. Internet-Drafts are working documents used by Internet standardizing groups like the Internet Engineering Task Force (IETF) to publish and discuss developments regarding the Internet.[3] An Internet-Draft may end up as an Request for Comments (RFC), if it is deemed appropriate. RFCs or Request for Comments are a series of technical and organizational documents concerning the Internet, including most Internet standards.[4] To find the most appropriate solutions for the FIDO2 TLS 1.3 Extension, this thesis discusses and enhances ideas brought up in other papers, like from Jonas Panizza, as well as brings in new ideas.

Additionally to writing the Internet-Draft, the existing POC implementation is changed to fit the changed and new ideas. To increase trust and understandability of the implementation, a dissector in the form of a Wireshark-Plugin is developed in tandem with the new PoC implementation. Wireshark is a software, that among other things, can be used to capture and analyze network packets.[5] Dissectors are used to analyze specific parts of a packet, in this case it makes the extension data of the FIDO2 TLS 1.3 Extension human-readable and structures it in a presentable form.

2. Internet-Draft "FIDO2 TLS 1.3 Extension"

The I-D "FIDO2 TLS 1.3 Extension" was created in tandem with this master's thesis, but is a self-contained document. The complete I-D can be found in appendix A. As individual documents, a PDF, HTML, plaintext and XML version of the I-D can be found on `https://github.com/wede-kind/I-D-FIDO2-TLS-1-3-Extension`.[6] This section describes what Internet-Drafts and Request for Comments (RFCs) are, how they are created and how they can be submitted. It also explains the specific choices made for the "FIDO2 TLS 1.3 Extension" I-D.

2.1. Internet-Drafts and Request for Comments

To understand what an Internet-Draft is, it is beneficial to first understand Request for Comments. An RFC is a technical or organizational document about the internet.[4] The collection of all RFCs is called the "RFC Series". The documents are produced by five streams: the IETF, the Internet Research Task Force (IRTF), the Internet Architecture Board (IAB), Independent Submissions, and Editorial. All Internet Standards-related documents are published as RFCs, but not all RFCs are Internet Standards-related documents. RFCs might also just be informational, experimental, historical, or Best Current Practice (BCP).[7] All RFCs start and are published as Internet-Drafts.

2.1.1. RFC Streams

- **Internet Engineering Task Force (IETF):**
  The IETF is a standards development organization, that creates voluntary standards, that may be adopted by internet users, network operators and equipment vendors. It focuses mainly on shorter term issues. It is also responsible for maintaining the Internet standards process, which includes the requirements for developing, reviewing and approving Standards Track and BCP RFCs. Anyone can participate in the IETF by signing up for a working group or registering for an IETF meeting.[3]

- **Internet Architecture Board (IAB):**
  The IAB is a board, that provides long-range technical direction for Internet development. The IAB's efforts are guided by fundamental design principles: The Internet's building blocks and their interactions. It currently has 13 individual members, that are selected by the IETF, as well as several ex-officio and liaison positions.[3]

- **Internet Research Task Force (IRTF):**
  The IRTF is an organization that focuses on longer term research issues related to the Internet. It consists of currently 15 research groups with stable, long-term memberships. Participation is by individual contributors.[8]

- **Independent Submissions:**
  Individuals can submit their Internet-Drafts to the Independent Submissions Editor (ISE) for possible publication as an RFC of the categories Informational, Experimental and Historic. Additionally to the standard review procedure for technical competence, relevance, and adequate writing, the submission is also reviewed by the ISE and by the IESG for possible conflict with the IETF process.[9]

- **Editorial:**
  For the publication of policies governing the RFC Series as a whole.[10]

The Internet-Draft "FIDO2 TLS 1.3 Extension" is treated, as if it is written for the Individual Submissions RFC Stream, even though it is not actually submitted.

2.2. Writing an RFC/Internet-Draft

When one wants to publish an RFC, one has to write an Internet-Draft first. There are multiple allowed authoring formats for writing an Internet-Draft. RFCXML, Markdown, other markup languages, plaintext, and Word processors like "Microsoft Word", that can convert their output into RFCXML.[11] For this I-D, RFCXML was used in combination with the editor Emacs, as this combination is widely used. It also comes with an easy-to-use tool, called "xml2rfc", for converting RFCXML into an Internet-Draft in the formats plaintext, HTML and PDF.[12]

Concerning language, style and structure, there exists an official RFC style guide[13]. It describes style conventions and editorial policies currently in use for the RFC Series. Concerning language, the I-D has to be written either completely in British English or American English. Among others, style choices concerning punctuation, capitalization, and abbreviation rules are also strictly defined and must be followed.
An Internet-Draft must include certain content:

- An **Abstract**

- A **Status of This Memo:**
  Status supplied by the RFC Editor.

- A **Copyright Notice**
- A **Summary of Changes:**
  Only written, if the Internet-Draft obsoletes or updates an existing RFC.
- **Security Considerations:**
  This has to be written to encourage document authors to consider security in their designs and to inform the reader of relevant security issues.[14]
- **Internet Assigned Numbers Authority (IANA) Considerations:**
  For registration of constants used in extensions of protocols to ensure they do not have conflicting uses and to encourage interoperability.[15]
- **References:**
  References in an RFC are divided into Normative and Informational references. Normative references include information that is necessary to understand or implement the technology in the new RFC, or include technology that must be present for the technology in the new RFC to work. Informative references only provide additional information and are not needed to implement the technology of the new RFC.[16]
- The **Authors' Addresses**

It is also encouraged to use diagrams to simplify complex concepts. The recommendation is to provide the diagrams both in ASCII-art and SVG. The SVG profile must, as of the writing of this thesis, conform with the "SVG 1.2 RFC" profile defined in RFC 7996.[17]

2.3. Submitting an Internet-Draft

Internet-Drafts are then submitted using the IETF Datatracker's submission tool.[18]

They have a fixed naming convention regarding the submitted file name. When using the independent submission stream, it must have one of the formats:

- draft-(author)-(subject)-(version)
- draft-(author)-(group)-(subject)-(version)

When using another stream, it must have one of the formats:

- draft-(source)-(group)-(subject)-(version)
- draft-(source)-(subject)-(version)

"source" is to be replaced by either the RFC stream, that adopted the I-D, or by the organization that authored it. "subject" describes the subject of the I-D in a few words. If a group adopted the I-D, "group" is to be replaced by the acronym of that group. "author" and "version" are self-explanatory. The characters that may be used in the name are restricted to the lowercase letters a-z, digits 0-9 and hyphens (-).

It is recommended to submit the Internet-Draft in XML format with a .xml file extension, but it might also be submitted as a text file with a .txt file extension.[11]

Following these conventions, the file name for the FIDO2 TLS 1.3 Extension I-D was chosen as `draft-fido2-tls-1-3-extension-00.xml`.

2.4. Content of the I-D "FIDO2 TLS 1.3 Extension"

Apart from the sections that need to be part of every Internet-Draft, the I-D includes optional sections, that are not specific to the FIDO2 TLS 1.3 Extension:

- Most Internet-Drafts have a section called Requirements Language or Conventions and Terminology, in which important keywords are defined. In this draft, this section is called `Conventions and Terminology`.
- A section called `Data Structures`, in which the composition of the structures described in the document is explained.
- The availability of one or more existing implementations of the protocol described in an Internet-Draft is considered an important matter. The existing Proof of Concept implementations are described in a section called `Implementation Status`.

The other sections in the I-D are specific to the FIDO2 TLS 1.3 Extension. These sections orientate themselves on the FIDO2 TLS 1.3 Extension described in section 5 in this master's thesis. Rather than describing all of these sections here, each subsection in section 5 indicated the equivalent section in the Internet-Draft. As the information in an Internet-Draft must be very precise and unambiguous, the sections in the Internet-Draft might contain more information than the equivalent section in this master's thesis, which may describe a concept more broadly.

3. Theoretical Framework

3.1. TLS 1.3

Transport Layer Security (TLS) and by extension TLS 1.3 aims to provide a secure channel for communicating peers. All information about TLS 1.3 can be found in RFC 8446.[19] The underlying transport has to be a reliable, in-order data stream. The secure TLS channel has to provide the following properties:

- **Authentication:** The server side has to be authenticated, the client side may be authenticated. Can happen via asymmetric or symmetric cryptography.
- **Confidentiality:** Data transmitted over the secure channel is only visible to the client and server after the connection is established. The length of the data is not hidden, but may be padded.
- **Integrity:** Data sent over the secure channel cannot be modified by attackers without detection.

TLS consists primarily of a handshake protocol and a record protocol. The handshake protocol is for establishing the TLS connection by authenticating the communicating parties, establishing symmetric session keys for encryption and other parameters. The record protocol handles the transmission, fragmentation, and protection of messages. As TLS extensions only directly interact with the handshake protocol, just the handshake protocol is explored further.

3.1.1. TLS 1.3 Handshake Protocol

The exact process of the handshake protocol is not needed to be understood, to fully understand the rest of the thesis, but certain key features are very important. These are further explained in a simplified explanation. The complete handshake protocol can be read about in RFC 8846.

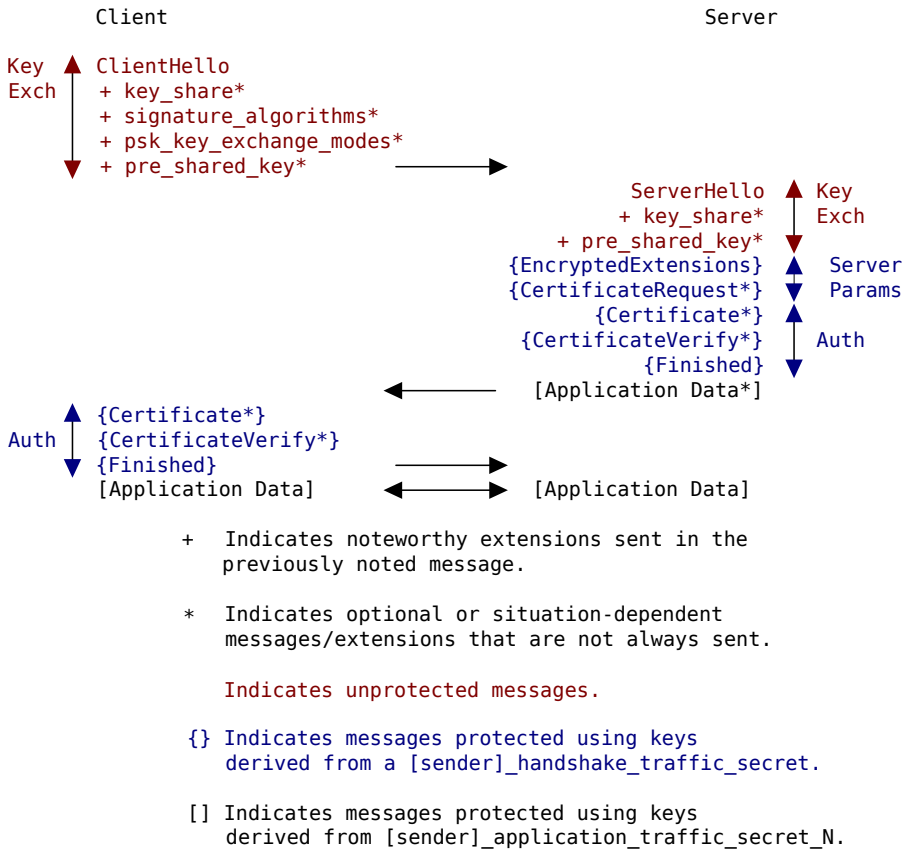

Figure 1: TLS Handshake. Altered version of Figure 1 in RFC 8446[19]

1. The Client initiates the handshake by sending a *ClientHello* to the Server. This includes many parameters, including key shares for the Key Exchange phase of the handshake. It may also include additional extensions. The *ClientHello* is not encrypted or authenticated.

2. The Server then processes the *ClientHello* and answers with a *ServerHello*, which indicates the negotiated connection parameters. These parameters include the server's side of the key share for the Key Exchange phase. The combination of the *ClientHello* and *ServerHello* determines the shared keys. The *ServerHello* is not encrypted. After the shared keys are established, the rest of the handshake is encrypted. Following the *ServerHello*, the Server may send an *EncryptedExtensions* message and a *CertificateRequest*. With the *EncryptedExtensions* message, the Server can respond to extensions included in the *ClientHello*. The *CertificateRequest* is sent, if certificate-based client authentication is desired. It may also include extensions. Lastly, the Server sends authentication messages to the server, including a *Certificate*, *CertificateVerify*, which is a signature over the entire handshake using

the private key associated with the *Certificate* and a *Finished* Message, which is a MAC over the whole handshake that is based on a session key derived from a shared secret.

3. The Client then responds with its own authentication messages, including a *Certificate* and *CertificateVerify* message, if previously requested with a *CertificateRequest*, and always a *Finished* Message. The *Certificate* may include extensions.

To simplify describing which part of the handshake is talked about, step 1 will further just be called "*Client Hello*", step 2 "*Server Hello*" and step 3 "*Client Authentication Messages*".

3.1.2. TLS Extensions

TLS 1.3 offers the possibility to extend the TLS handshake via extensions. Extensions are sent in the following structure:

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
```

The `extension_type` identifies the extension. A list of registered extension types is registered with the IANA.[20]
The `extension_data` contains data specific to the extension and message.

Extension messages are not allowed to be sent in every part of the TLS Handshake, but generally follows a predefined request/response system. In some cases, a response is not necessary. The Client sends its extension requests in the *ClientHello*. The Server may answer these request in the *ServerHello*, *ExtendedExtensions* and *CertificateRequest*. The server can send extension requests in the *CertificateRequest* message, which the client may answer in a *Certificate* message. Sending extension responses without the corresponding request is not allowed. Extension messages in the *ClientHello* are not encrypted. The extensions sent in the *ServerHello* are not encrypted, while the *EncryptedExtensions* and *CertificateRequest* in the *Server Hello* are already encrypted using the `server_handshake_traffic_secret` The extensions in the *Certificate* messages are encrypted, as the complete *Client Authentication Messages* are encrypted. Extension messages are allowed to carry arbitrary data in their `extension_data`, which may be independent of where in the handshake they are sent.[19]

3.2. Fast IDentity Online 2

Fast IDentity Online (FIDO) is a project of the FIDO Alliance, that allows easy passwordless authentication over the web. The FIDO Alliance is an open industry association, whose goal is to reduce the worlds' reliance on passwords. Many companies and governments are part of the FIDO Alliance, including Germany with the Federal Office of Information Security. FIDO uses authentication credentials, that use public-private key pairs to authenticate end-user devices with websites or apps. No secrets are shared when authenticating, which makes FIDO phishing-resistant by design. The authenticators can be either external hardware tokens, also called security keys, for example USB, Bluetooth or NFC, or they can be directly integrated into the platform running the software, for example Windows or Android. These authenticators are called platform authenticators. As the authenticators provide immediate proof of identity, they are usually further protected by steps like biometrics, PINs, or patterns. FIDO2 is the newest version of FIDO as of publishing of this document. It consists of the two protocols Client to Authenticator Protocol (CTAP) 2.2[21] and Web Authentication (WebAuthn) Level 2[22]. CTAP is used for communication between security keys and client platforms.[1] For the purpose of this thesis, it is assumed that CTAP 2.2 is a working and secure communication channel. The exact functionality is not further important in the context of this thesis. WebAuthn plays a major role in the design of the FIDO2 TLS 1.3 extension, which is why WebAuthn is further explored in a separate section.

3.3. Web Authentication Level 2

WebAuthn is an Application Programming Interface (API) enabling the creation and use of public key-based credentials for web applications. The first iteration of Web Authentication, Level 1, was created by the Word Wide Web Consortium (W3C) in 2019. The newest version as of writing of this thesis is the W3C recommendation "Web Authentication: An API for accessing Public Key Credentials Level 2" from 2021.[22] The messages defined for the FIDO2 TLS 1.3 Extension are mostly modeled after the API defined in this standard.

The idea of the API is based on, that the credentials belong to the user and are managed by a WebAuthn Authenticator. A WebAuthn Authenticator is a cryptographic entity, that can exist in hardware or software. They can register a user with a Relying Party (RP), assert possession of the registered public key credential, and optionally verify the user. The WebAuthn Relying Party is the entity, that uses the API to register and authenticate users through the users clients. In the context of WebAuthn, the RP is

most often a website.

The processes for registering or authenticating a user with a Relying Party are called ceremonies. These ceremonies are further described in a simplified form, because the exact procedure is not necessary to understand the FIDO2 Extension. The main focus is put on the message flow between the client and the Relying Party and the exact contents of each message, which is specifically marked bold like `this` in the descriptions, as these are the values that are relevant for the messages of the FIDO2 TLS 1.3 Extension. The exact description of the ceremonies can be found in the WebAuthn, Section 7.

3.3.1. WebAuthn Registration

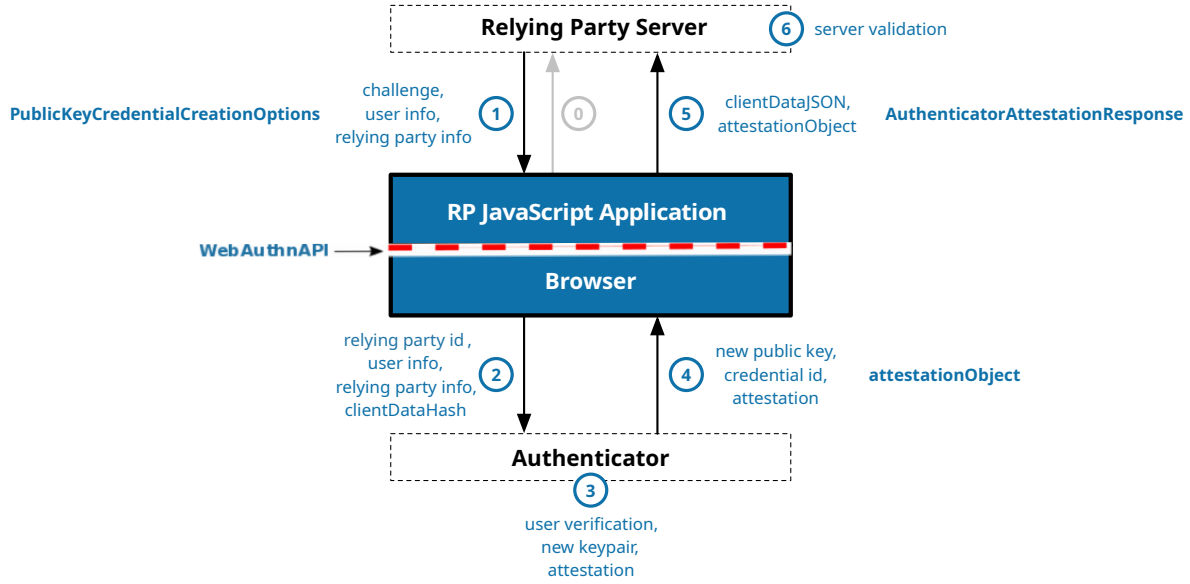

Figure 2: WebAuthn Registration Ceremony[22]

Figure 2 shows the steps having to be done to register a new credential with a RP.

0. In step 0 shown in the figure, the user starts the registration progress by interacting with the Relying Party via a user agent, for example by clicking a button on a website. The most common user agents are browsers. The user has to provide a `user display name` to the Relying Party. Additionally, there may be an additional authentication step necessary, if the RP does not allow every user to register a new credential, or wants to connect it to an already existing account. This additional authentication step may be done using already existing FIDO credentials, or traditional authentication methods like passwords or E-Mail.

1. If the preliminary authentication steps are successful, the RP uses the information received by the client, in addition with RP configured data to create a `PublicKeyCredentialCreationOptions` object, further just called *options*. This object contains the `RP entity`, including the `RPs id and name`, the `user entity`, including a `user name` provided by the RP and `user display name` provided by the user and a unique `user ID` created by the RP. The other mandatory values are a `challenge` for the authenticator and a list of public key credential parameters called `pubKeyCredParams`, which contains information about the desired properties of the credential that is to be created. The *options* object also includes the optional values `timeout`, `excludeCredentials` to limit creating multiple credentials for the same account on the same authenticator, the `authenticatorSelection`, which lists the criteria, the RP has for the authenticators allowed in the registration and possible extensions, `attestation`, which shows the RPs preference for attestation conveyance and possible `extensions`. The RP then sends the *options* object to the client and calls a *create* function, which takes the RPs `origin`, `cross-origin` and *options* object as arguments.

2. When the *create* function is called, the client creates a new `collectedClientData` object, which includes the `type` "webauthn.create", a base64url-encoding of the `challenge` from the *options* object, the `origin`, `crossOrigin` and an optional value called `tokenBinding`. It then creates a JavaScript Object Notation (JSON)-serialized version of the `collectedClientData` called `clientDataJSON` and a SHA256-hash of the JSON-serialized version called `clientDataHash`. The client then calls the authenticator using the *authenticatorMakeCredential* function and passes the `clientDataHash`, a `requireResidentKey` boolean, which describes the resident key requirement for credential creation, `requireUserVerification` boolean and along with data from the *options* object from the RP to the authenticator.

3. After receiving a *authenticatorMakeCredential*, the authenticator creates a new credential including a public/private key pair. User presence and verification is checked via the touch of a button, PIN entry or other methods. If the new credential is discoverable, its private key along with information for later identification, including the `RP ID` and `user ID`, is completely stored on the authenticator itself. If it is a server-side credential, also called non-discoverable credential, only a part of it or nothing at all is stored on the device and relevant information is sent to the RP. The authenticator uses a master secret to regenerate the complete credential

when presented with the appropriate information.

4. An `attestationObject` is created by the authenticator and after creating a new credential and sent to the client. It includes `authenticator data` and an `attestation statement`, which is both cryptographically protected against and opaque to the client, as well as additional information, that the RP may use to validate the attestation statement and to decode and validate the authenticator data. This `attestationObject` is then passed to the client by the authenticator.

5. The client then creates an *AuthenticatorAttestationResponse* object. This object contains the `attestationObject` and the `clientDataJSON` created by the client in step 2. The *AuthenticatorAttestationResponse* is then sent to the RP.

6. After receiving the *AuthenticatorAttestationResponse*, the Relying Party does verification steps on the `attestationObject` and the `clientDataJSON` to ensure, that the credential and user information are valid. If all verification steps are successful, the RP registers the new credential with the account denoted in the user information of the *options* structure.

3.3.2. WebAuthn Authentication

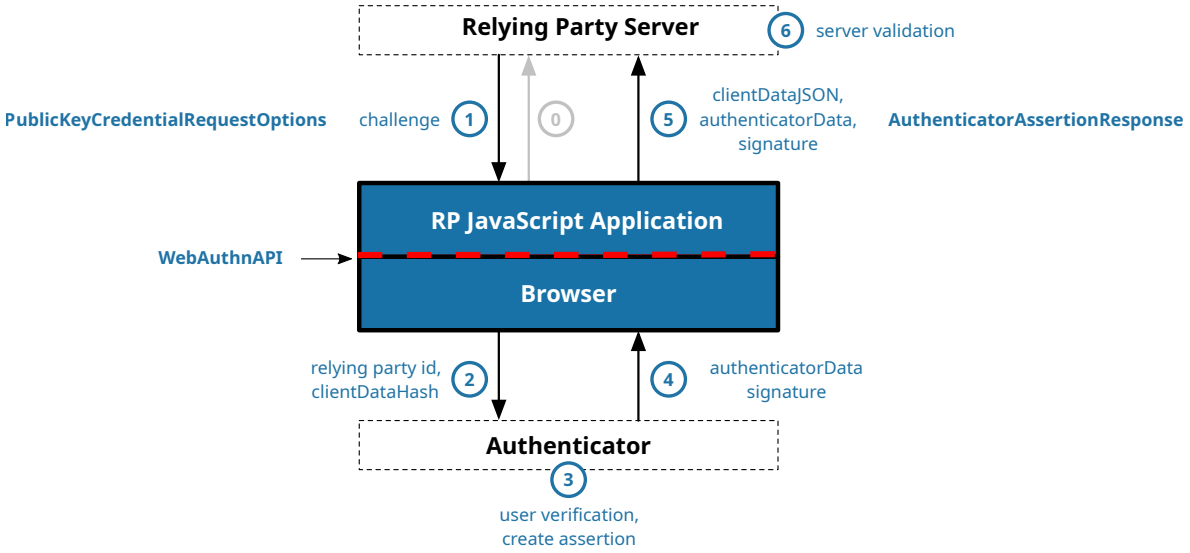

Figure 3: WebAuthn Authentication Ceremony[22]

Figure 3 shows the steps for authenticating an existing registered credential with a Relying Party.

0. The user starts the authentication by trying to access a resource with the RP that requires authentication. For example logging into a user account. The client sends an authentication request to the RP.

1. When the RP receives an authentication request, it creates a `PublicKeyCredentialRequestOptions` object configured to the needs of the upcoming ceremony, which will further be called *options*. This structure includes a mandatory `challenge` and multiple optional values. These optional values are a `timeout`, the `RP ID`, `allowedCredentials`, representing public key credentials acceptable to the caller, mandatory if server-side credentials are used, an `userVerification` member, specifying the RPs requirements regarding user verification, and `extensions`. To create the `allowedCredentials`, the RP has to know the identity of the user, which has to be provided by the user using classic web mechanics. The *options* object is then sent to the client by the RP and calls a *get* function, which takes only the *options* as an argument.

2. When the *get* function is called, the client creates a new `collectedClientData` object, which includes the `type` "webauthn.create", a base64url-encoding of the `challenge` from the *options*, the `origin`, `crossOrigin` and an optional value called `tokenBinding`. It then creates a JSON-serialized version of the `collectedClientData` called `clientDataJSON` and a SHA256-hash of the JSON-serialized version called `clientDataHash`. The client then calls the authenticator using the *authenticatorGetAssertion* function and passes the `clientDataHash`, `RP ID`, `requireUserPresence` boolean, `requireUserVerification` boolean and coming from the RP, taken by the client out of the *options* structure, the `allowedCredentials` and `extensions`.

3. After the *authenticatorGetAssertion* is called, the authenticator determines a fitting credential if possible. If no `allowedCredentials` are specified, the credential has to be a discoverable credential, so the authenticator has to identify it itself using the `RP ID` and `user ID` as keys. If `allowedCredentials` are specified, which must be the case for server-side credentials, the authenticator identifies the credential by searching it with the supplied credential IDs or regenerates it with the help of its master secret. After identifying an appropriate credential, user presence and verification is checked via the touch of a button, PIN entry or other methods.

4. The authenticator then uses the identified credential to creates an `AuthenticatorResponse` which includes `authenticatorData` consisting of data

collected in the authentication and a `signature`, which is created by signing the concatenation of the `authenticatorData` and `clientDataHash` with the private key of the credential. It may also include the `userHandle`, equivalent to the `userID`, of the selected credential. The `AuthenticatorResponse` response is then sent to the client.

5. The client then builds an object called `AuthenticatorAssertionResponse`, which includes all elements of the `AuthenticatorResponse` and the `clientDataJSON` created by the client in step 2. This `AuthenticatorAssertionResponse` is then sent to the RP.

6. Finally, the RP does various verification steps on the received `AuthenticatorAssertionResponse` that, if successful, complete the authentication.

4. Related Work

4.1. EAP-FIDO

"EAP-FIDO" is an Internet-Draft that aims to integrate FIDO into the Extensible Authentication Protocol (EAP), a widely used standard that allows a server to authenticate a client using different authentication methods. It was released by Jan-Frederik Rieckers and Stefan Winter in 2023, with the newest version being from September 2025. According to the I-D, common EAP-methods use EAP-TLS to provide confidentiality of the inner authentication, which is mostly password based. As the inner authentication is not protected, this could be a possible flaw, if the client has no knowledge of the expected server name, which is for example the case in eduroam, as attackers impersonating the server could then observe the password. With FIDO, even if an attacker compromises the TLS connection, it does not get access to critical information, as the private key is never revealed. In the overall protocol-design of EAP-FIDO, TLS is used to authenticate the EAP-FIDO server to the client. The established TLS connection is then used to authenticate the client via a traditional FIDO ceremony. Client and server can then implicitly derive keying material from the TLS handshake. EAP-FIDO is different to the FIDO2 TLS 1.3 Extension in that it does not perform the FIDO ceremony directly in the handshake, but in an established EAP-TLS tunnel instead. Also, it only specifies authentication, while registration is out-of-scope and has to still be done using traditional methods. Nevertheless, it shares some important questions, for example how to determine the Relying Party ID.[23]

4.2. FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks

In his master's thesis "FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks", published in 2024, Jonas Panizza proposes a TLS 1.3 Extension designed to integrate the FIDO2 authentication and registration ceremonies directly into the TLS handshake. The goal is to bring FIDO2 to non-HTTPS applications. The thesis explores a "double handshake" as a viable option to integrate FIDO2 into only the handshake messages. Problems that arise by making FIDO available outside the web context are also tackled. This includes a new control mechanism for key registration, encoding of the messages, determination of the origin and Relying Party ID, as well as exploring solutions for communicating a "Finished" message, which does not fit into the double handshake. The extension is implemented for OpenSSL in C as a Proof of Concept. The practicality of the extension is shown by using it to establish an 802.1X EAP-TLS Wi-Fi network that uses FIDO hardware authenticators.[2] Jonas Panizza's thesis builds upon further groundwork by Tom-Lukas Johann Breitkopf[24] and Mario Freund[25], which is not further explored. This master's thesis uses the research by Jonas Panizza as a basis for the definition of the FIDO2 TLS 1.3 Extension.

5. FIDO2 TLS 1.3 Extension

This section describes the FIDO2 TLS 1.3 Extension, further also just called "FIDO2 Extension", and explains the design choices leading to its design. The FIDO2 Extension predominantly deals with the transportation of information between the client and the Relying Party in the registration and authentication ceremonies. Additionally, as the extension disconnects FIDO from the web context required by WebAuthn, some changes have to also be made to the processing of data by the client and RP. For steps that cannot be easily translated into TLS, new solutions have to be found. An extension was chosen over integrating FIDO into the TLS handshake itself, because changing the TLS protocol itself has big security and complexity implications.

As TLS and FIDO2 both use the concepts of a client and server, they are redefined for the context of the FIDO2 TLS 1.3 Extension:

- **Client:** Consolidates the functionality of a WebAuthn client and a TLS client.
- **Server:** Consolidates the functionality of a FIDO server incorporating the Relying Party and a TLS server.

In WebAuthn, two types of credentials are specified. Discoverable credentials and server-side credentials as defined in section 3.3.1 step 3. To make authentication with server-side credentials possible, the RP has to have knowledge of the user's identity, while this is not necessary for authentication with discoverable credentials. For this reason, in the context of the FIDO2 Extension, the authentication with discoverable credentials and authentication with server-side credentials are seen as separate ceremonies.

5.1. Integration into the TLS Handshake

The communication happening between the client and the server in WebAuthn can be classified into three steps for both registration and authentication. An *Indication* from the client to the server, *Request* from the server to the client and *Response* again from the client to the server. Respectively steps 1, 2 and 5 in sections 3.3.1 and 3.3.2. As established in section 3.1.2, the client and server can send each other arbitrary extension data in the *Client Hello*, *Server Hello* and the *Client Authentication Messages*, when client certificates are used. As the second communication from the client to the server is always needed, the FIDO2 TLS 1.3 Extension has to use the extension mechanism of the client certificates. To ensure the client has always access to a valid certificate, the usage of a dummy certificate is proposed. This dummy certificate does not have to

actually authenticate the client, but is only used to transport the extension data. To make the dummy certificate easily recognizable, it may be given unique characteristics. Referred to in the Internet-Draft in appendix A, section 10. Translating the WebAuthn steps to TLS 1.3, the *Indication* can be sent in the `ClientHello`, the *Request* in the `CertificateRequest` and the *Response* in the `Certificate` sent by the client. An important observation is, that in the ceremonies for registration in 3.3.1 and authentication with server-side credentials in 3.3.2, information about the client is communicated in the *Indication* that must remain secret to untrusted third parties. As the extension data sent in the `CLientHello` is not protected by the TLS transport protection, it has to be additionally encrypted. To make encryption and decryption of that data possible, a key must have to be exchanged between the client and server before the `ClientHello`, which is not possible in a single TLS handshake. A possible solution for this problem, while still purely using the unmodified TLS 1.3 handshake mechanism, is to do a double handshake, which is discussed in section 5.3. The encryption is explained in detail in 5.4

For the last type of ceremony, authentication with discoverable credentials, no secret data has to be communicated in the *Indication*, so a single handshake is sufficient, which is discussed in the next section 5.2. The integration into the TLS handshake is described in the I-D in appendix A, sections 6 and 11.

Out-of-scope of WebAuthn, but still often part of the FIDO ceremony, is the communication of the *Result* of the ceremony. This would need an additional second message from the server to the client, which a TLS 1.3 handshake does not provide. A solution for the *Result* message is discussed later in section 5.9.

5.2. Single Handshake

In the single handshake approach, all data exchanged between the client and server is communicated in the extension data of the **ClientHello**, **CertificateRequest** and **Certificate** sent by the client. As a single handshake is only possible for authentication with discoverable credentials, it is presented with the specifics of that ceremony.

5.2.1. Handshake for Authentication with discoverable credentials

The authentication with discoverable credentials is the least complicated ceremony, as the client does not have to send secret information in the *Indication*, so a double handshake is not needed.

Authentication with discoverable credentials integrated into a single TLS 1.3 handshake looks as following. More information about the protocol flow and message specifications can be found in the I-D in appendix A, sections 11.4, 12.6, 12.8 and 12.9.

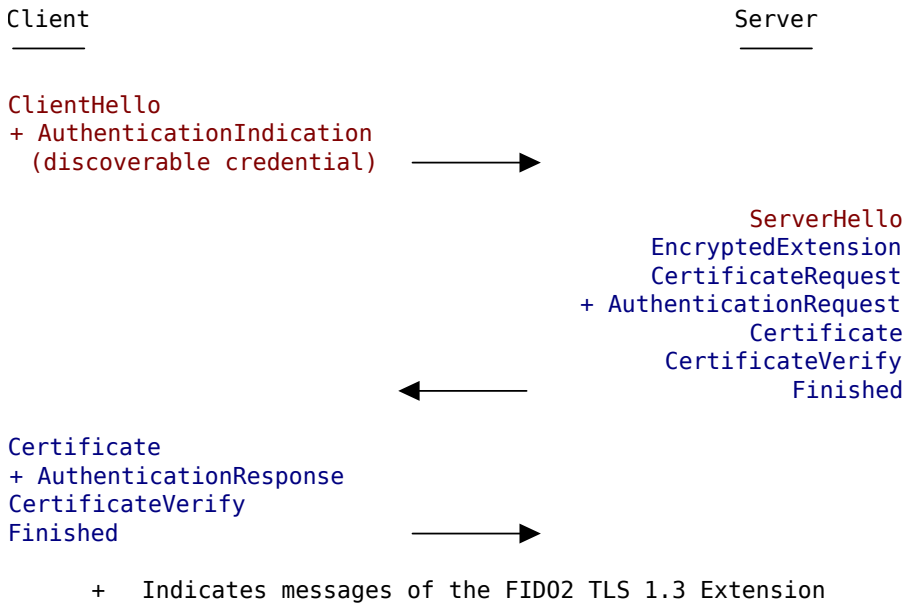

Figure 4: Authentication Handshake (discoverable credential). Graphic created by author.

1. **Authentication Indication (discoverable credential):**
   Includes only a `Message Type`, which is an identifier used for easier message processing and debugging. In all *Indications*, it also serves the RP to identify the client's desired action.

2. **Authentication Request:**
   Includes a `Message Type` and the values from WebAuthn specified in section 3.3.2 step 1, explicitly without `allowedCredentials`.

3. **Authentication Response:**
   Includes a `Message Type` and the values from WebAuthn specified in section 3.3.2 step 5. Must include the `UserHandle`.

5.3. Double Handshake

For the registration and authentication with server-side credentials, a single TLS handshake is not sufficient. The reason is, that in these cases, the client needs to send sensitive data to the server in the *Indication*. As the server is only able to send data once in

the `CertificateRequest` in a single handshake and the `ClientHello` message is not encrypted, a double handshake is needed. The idea of using a double handshake for protecting sensitive data was already used for instance in RFC 4680[26], although in a different form.

The sole purpose of the first handshake, called "PreHandshake", is to enable encryption in the second handshake. The second handshake includes the data exchanged for the actual ceremonies. That means the exact same PreHandshake can be used for both registration and authentication with server-side credentials. There are two values distributed in the PreHandshake. The first value enables the server to connect the second handshake to the first handshake and is called `ephemeral user ID`. The second value is the `key` used by the client to encrypt the client specific data in the `ClientHello` of the second handshake, which is explained in 5.4.

The PreHandshake looks as following. More information about the protocol flow and message specifications can be found in the I-D in appendix A, sections 11.2, 12.1 and 12.2.

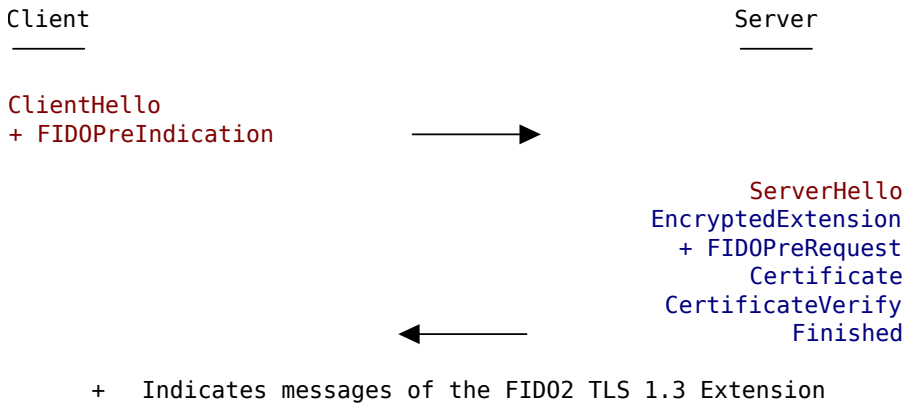

Figure 5: PreHandshake. Graphic created by author.

1. **PreIndication:**
   Includes only a `Message Type`.

2. **PreResponse:**
   Includes a `Message Type`, `ephemeral user ID` and `256-bit key`.

After the Client receives the PreResponse, it may open up a new, or reuse the same Transmission Control Protocol (TCP) connection and start the second handshake. The *Client Authentication Messages* are not part of the PreHandshake, as it does not have to

be finished for the double handshake to work. The handshake may be finished, but no extension data has to be sent in the `Certificate`.

Doing a double handshake like this, by sending an unencrypted `ephemeral user ID` in the `ClientHello` of the second handshake, opens a new attack vector for a possible Man-in-the-middle (MITM) attack. Sensitive client data is also sent in the *Request* of the second handshake, which is sent in the `CertificateRequest` of the *Server Hello*. In a single handshake, this is no problem, as this data is protected by the TLS transport protection. In a double handshake, the MITM attacker could intercept the `ClientHello` of the second handshake and then start an own TLS handshake replaying the extension data of the `ClientHello` to get access to all sensitive data sent in the *Server Hello*, as the server could not distinguish the attacker from the legitimate user. For this reason, the sensitive data in the *Server Hello* has to be also encrypted with the key exchanged in the PreHandshake, explained in 5.4. In the thesis of Jonas Panizza, the reason to exchange a key for symmetric encryption is purely to counteract this MITM attack. In the version of the double handshake described in his thesis, the sensitive data of the `ClientHello` in the second handshake here is instead transported in the *Client Authentication Messages* in the PreHandshake. This requires two different PreHandshakes for registration and authentication with server-side credentials. As encryption has to be used anyway to prevent the MITM attack, the PreHandshake can be simplified as described above. A generalized version of the double handshake can be found in the I-D in the appendix A, section 6.

As both registration and authentication with non-discoverable credentials use the double handshake mechanism, their second handshake is described separately.

5.3.1. Handshake for Registration

The registration handshake has to be preceded by a PreHandshake.
The second handshake for registration looks as following. More information about the protocol flow and message specifications can be found in the I-D in appendix A, sections 11.3, 12.3, 12.4 and 12.5.

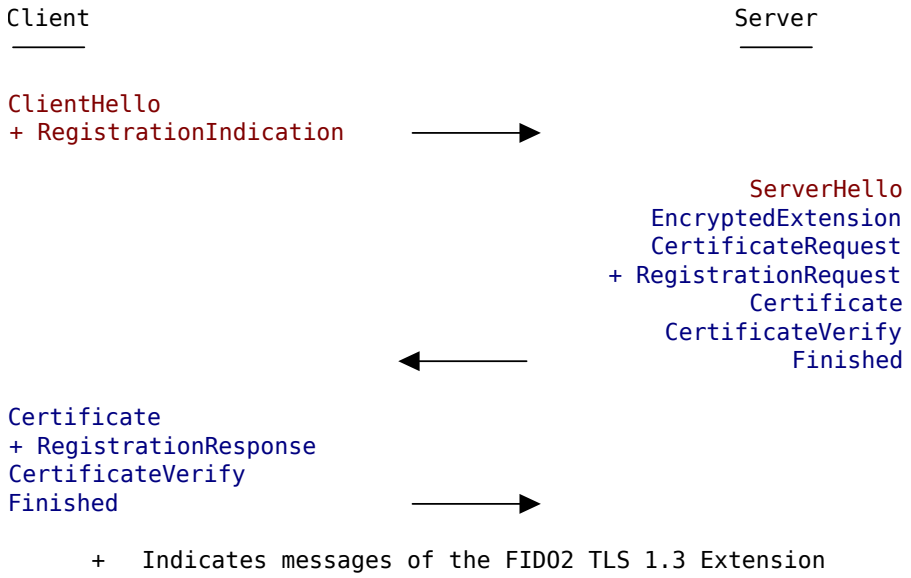

Figure 6: Registration Handshake. Graphic created by author.

1. **Registration Indication:**
   Includes a `Message Type`, `ephemeral user ID` and a `ticket`, which is later explained in section 5.6. From WebAuthn specified in section 3.3.1 step 0 it also includes a `user display name`. The `ticket` and `user display name` are encrypted with the 256-bit key from the PreHandshake.

2. **Registration Request:**
   Includes a `Message Type` and the values from WebAuthn specified in section 3.3.1 step 1. The client specific data is encrypted with the 256-bit key from the PreHandshake.

3. **Registration Response:**
   Includes a `Message Type` and the values from WebAuthn specified in section3.3.1, step 5.

5.3.2. Handshake for Authentication with server-side credentials

The handshake for authentication with server-side credentials has to be preceded by a PreHandshake.
The second handshake for authentication with server-side credentials looks as following. More information about the protocol flow and message specifications can be found in the I-D in appendix A, sections 11.4, 12.6, 12.8 and 12.9.

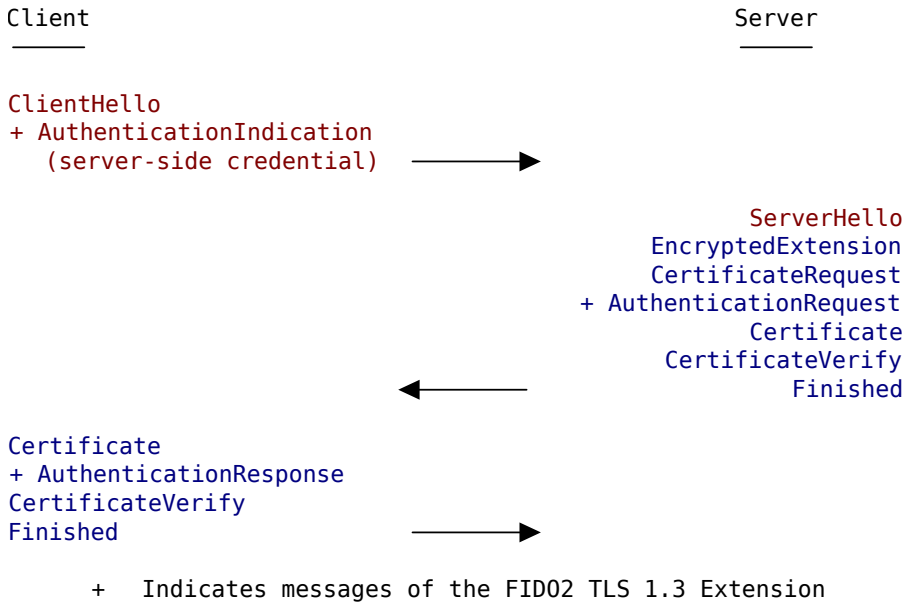

Figure 7: Authentication Handshake (server-side credential). Graphic created by author.

1. **Authentication Indication (server-side credential):**
   Includes a `Message Type` and `ephemeral user ID`. Additionally, as the RP has to identify the user, it includes a `user name`, which is encrypted with the 256-bit key from the PreHandshake.

2. **Authentication Request:**
   Includes a `Message Type` and the values from WebAuthn specified in section 3.3.2 step 1. Must include the `allowedCredentials`, which are encrypted with the 256-bit key from the PreHandshake.

3. **Authentication Response:**
   Includes a `Message Type` and the values from WebAuthn specified in section 3.3.2 step 5. Including the `UserHandle` is optional.

5.4. Encryption

In the case of registration and authentication with server-side credentials, sensitive client information has to be sent in the extension data sent in the `ClientHello`. As the `ClientHello` is not protected by the transport protection of TLS, the data has to be encrypted. Additionally, With the introduction of the double handshake, sensitive client information has to also be protected in the `CertificateRequest`, to prevent the MITM attack described in section 5.3.

There are multiple ways, in which such an encryption can be accomplished, but there are certain criteria that should be fulfilled in case of encryption for TLS:

- The encryption has to be easily accessible for all clients,
- it should be forward secure,
- and with the introduction of the double handshake, a new requirement arises, as the extension has to work well with load balancers. There is no guarantee, that the client will connect with the same physical server in both handshakes, if the second handshake happens over a new TCP connection.

To make the encryption as accessible as possible, the idea is to use an encryption algorithm that is also needed for another part of the normal TLS 1.3 handshake. RFC 8446, which describes TLS 1.3, has a section about mandatory-to-implement cypher suites. It states, that Advanced Encryption Standard (AES) Galois/Counter Mode (GCM), specifically `TLS_AES_128_GCM_SHA256` must be implemented by TLS-compliant applications and `TLS_AES_256_GCM_SHA384` is heavily encouraged to be implemented.[19] The latter is a better choice, because it is cryptographically stronger and should be implemented by almost all platforms as of writing of this thesis. The encryption method included in the `TLS_AES_256_GCM_SHA384` cypher suite is `AEAD_AES_256_GCM`, further also just called `AES-256-GCM`, which is explained in RFC 5116.[27] This fits the accessibility requirements, as the client and server have to support TLS 1.3 anyway. Additionally to encryption, `AES-256-GCM` also provides authentication of the encrypted data, which is useful for the data encrypted in the `ClientHello`, as that is not authenticated by TLS.

`AES-256-GCM` uses *256-bit encryption keys*. There are multiple approaches on how this key can be generated by the server and distributed to the client.

The simplest approach and the approach used by Jonas Panizza is, that the server generates both a random *ephemeral user ID* and a random *256-bit key* and then sends them to the client in the `ServerHello` of the PreHandshake. The server has to save both the *ephemeral user ID* and the *256-bit key* while waiting for the second handshake. They should not be held longer than a few minutes, as the server opens itself to Denial-of-service (DoS) attacks otherwise. If a client has problems to do the second handshake in time, they can just redo the PreHandshake. To decrease the message size, the server could only randomly generate the *256-bit key*, with the *ephemeral user*

ID being the hash of the key. This makes it possible to only send the *256-bit key* to the client, because it can calculate the *ephemeral user ID* themselves. Because of the one-way property of cryptographic hash functions, this creates no further security risk at the moment of communication, but can invalidate forward security, if the hash function is broken. This variant has problems with load balancers, as the *ephemeral user ID* and *256-bit key* have to be saved in a central database or distributed to all physical servers before a second handshake can be made reliably with every physical server.

In a second variant, the *ephemeral user ID* and *256-bit key* are not calculated by only the server, but rather derived from the TLS session-key of the PreHandshake. In this case, the payload of the extension data in the PreHandshake can be empty apart from the `message_type` fields. The client and server then have to save the *ephemeral user ID* and *256-bit key* for usage in the second handshake. This variant has the advantage of being guaranteed forward secure, as the session key is temporary by design. For usage with load balancers, this variant has the same problem as the first one.

To combat the problem with the load balancers, another solution is needed, in which the server needs to hold no session or user specific state between the two handshakes. As the only information the server gets from the client in the second handshake, that can be used to connect it to the first handshake, is the *ephemeral user ID*, it has to be part of the solution. In this solution, the server generates a random *ephemeral user ID* and then derives the *256-bit key*, using a master secret only the server knows by calculating

`256_bit_key = SHA256(ephemeral_user_id||master_secret)`.

It then sends the *ephemeral user ID* and *256-bit key* to the client. The server does not save any state. When the client starts the second handshake and sends the *ephemeral user ID*, the server can then use its master secret to generate the *256-bit key* again. This solves the problem with load balancers, as a master secret is easy to share over all physical servers. The problem is, that this variant has no forward security. Any entity getting hold of the master secret can decrypt the extension data of every handshake previously recorded. To counteract this, the master secret can be renewed every few minutes, with the old key being permanently deleted. As second handshakes could fail, because the master secret is switched right after the PreHandshake, the server could hold both the current master secret and the second newest. The server could then try the current master secret first and if that does not work also the master secret one iteration old. To find out if a master secret is correct, the server can verify the GCM-Tag of the encrypted data.

To simplify the encryption process as much as possible, if multiple values that need to be encrypted are part of a single message, they are combined into one `encrypted_data` object, which is a CBOR array that is then encrypted as a whole. This concerns the *Registration Indication* and *Registration Request*.

Using pure `AES-256-GCM` for encryption creates a new attack vector, because the length of the input can be directly derived from the length of the output. This makes it possible to get the length of the `user name` and `user display name` in registrations and just the `user name` when authenticating with server-side credentials, thereby making it possible to potentially identify certain users. To prevent this, the `user name` and `user display name` have to be padded to obscure the length of the unencrypted data. Simple bit padding is sufficient. When using bit padding in this context, one '1' bit followed by as many '0' bits as necessary to meet a predefined length of 256 bytes is added to the `user name` and `user display name`. This increases the size of the payload of the *ClientHello* in the second handshake, but the size increase is low enough, that this introduces no further problems. The corresponding section in the I-D in appendix A is section 8.

5.5. Message Encoding

The data that is transferred in the extension data should be encoded in a way, that is both space efficient and flexible, while also being accessible for every possible client that wants to use the extension. TLS is size constraining, as it automatically splits up TLS records into multiple records, if a single record gets too big, with a maximum size of 2^{14} bytes per record.[19] The splitting of records wants to be avoided if possible for efficiency reasons. The encoding has to be flexible, as the FIDO2 Extension has optional values in its messages.

To make sure that the client has access to the encoding method, it is useful to look at what encoding methods are already used for TLS 1.3 and mandatory parts of FIDO2, as these will have to be supported anyway. TLS uses Abstract Syntax Notation One (ASN.1) for encoding of specific elements like X.509 certificates. WebAuthn uses JSON objects to encode elements like the `ClientDataJSON`, which is then also encoded in JSON in the FIDO2 Extension. CTAP uses Concise Binary Object Representation (CBOR) for communication between the client and the authenticator. This means JSON, ASN.1 and CBOR are all possible encodings.

JSON objects are very flexible and human-readable, but sacrifice space-efficiency for that. ASN.1 is space efficient, but requires pre-defined schemas, which are a problem as the extension messages incorporate optional data. CBOR is both very space efficient and allows the usage of maps for the optional data. Additionally, using CBOR removes complexity and a potential source of error, as the data does not have to be recoded from JSON to CBOR by the client before being sent to the authenticator, like in WebAuthn. This makes CBOR, as also pointed out by Jonas Panizza the best choice for the FIDO2 TLS 1.3 extension.[2] For debugging purposes, it can also be made human-readable by using dissectors for example. Such a dissector will be later discussed in section 6.2.

Consequently, the messages in the FIDO2 Extension are encoded in a CBOR array, with the optional values being encoded in a CBOR map. Unsigned integers are used instead of text strings for keys to save space. CBOR does not support enums directly, but small numbers can still be encoded in one byte. Tuples are stored in CBOR arrays, while optional values with arbitrary many occurrences are organized in nested CBOR arrays. To learn more about the sizes of messages using the CBOR encoding, information about that can be found in the thesis of Jonas Panizza.[2] The exact definition of every message defined by the extension can be found in the Internet-Draft accompanying this thesis found in section 12 of the I-D found in appendix A.

5.6. Pre Registration Authentication

As described in step 0 of Section 3.3.1, the original FIDO registration may use additional authentication steps before permitting a user to register a new credential. In traditional FIDO2, this is normally done using other authentication methods like passwords or email verification. This interaction mostly happens over a website in the browser. As this interaction is omitted by registration over TLS, a new way to authenticate the client prior to registration is needed.

One possibility would be to use already existing authentication methods supported by TLS 1.3, like Client Certificate Authentication, Kerberos, TLS-Secure Remote Password, TLS-OpenID or TLS-OAuth. After this authentication concludes, the server could either continue the registration, or cancel it. One problem is, that these authentication methods generally are only concluded at the end of the handshake, which means that the client first notices, that it is not allowed to register with the RP, after it has already completed the whole registration ceremony. If a complete PreHandshake would be made,

including the Client Authentication messages, the RP could already verify if the client is allowed to register, before the second handshake and then cancel the ceremony already after receiving the Client Hello with the corresponding ephemeral user ID. As the Client Authentication messages are to be generally omitted in the PreHandshake to simplify it, this would make the PreHandshake more complicated and additionally would require a different PreHandshake for registration and authentication with server-side credentials again, which is not desirable.

Another solution proposed by Jonas Panizza is to use a pre shared secret called a **"ticket"**. This `ticket` has to be known by both parties before the actual registration ceremony and would therefore be distributed from the Relying Party to the client out of band. It is then shared with the RP in the **Registration Indication**, which can validate the `ticket` with its database.[2] The `ticket` was chosen to be a 256 bit randomized string. There is also the possibility to give the `ticket` extra semantic. It could for example include an expiration date or information about the user. This would save the server extra resources, as it would then not have to save this data separately. As the `ticket` has to be encrypted anyway when being sent to the server, this would also not expose any sensitive data. To ensure, that no `ticket` is accidentally used twice, it must still have a random component of at least 128 bits. As the semantic adds complexity to an otherwise very simple mechanism, it is not recommended, but also not strictly forbidden. The ticket mechanism was adopted for the Internet-Draft, as it was deemed a solution, that adds the least complexity to the registration ceremony, while still providing a functioning authentication mechanism. The corresponding section in the I-D in appendix A is section 9.

5.7. User Name and User Display Name

In WebAuthn, the `user display name` should be chosen by the user and the `user name` by the server, but might each also be chosen by the other party.[22] This exchange happens through an additional dialog via HTTPS. As this dialog cannot happen in case of the FIDO2 Extension, it is strictly defined, that the user always choses the `user display name` and the server always choses the `user name`. Additionally, as the `user name` has to be used to identify the user in case of authentication with server-side credentials, it must be chosen unique among all `user names` registered with the server. If the uniqueness criterion is fulfilled, the `user name` might be chosen as equal to the `user display name`.

5.8. Origin and RP ID

FIDO uses a concept called `origin` to make sure that the client communicates with a trusted server.[22] The origin is described in the HTML Living Standard[28]. There are two types of origins. Opaque origins, that are only used in special cases and Tuple origins. For Webauthn, only tuple origins are relevant. A tuple origin consists of:

- a **scheme:** an ASCII string (for example “HTTPS”).
- a **host:** a domain, an IP address, an opaque host, or an empty host.
- a **port:** null or a 16-bit unsigned integer.
- a **domain:** null or a domain, typically null.

As this definition of an `origin` was developed for the web, it is not easily adoptable in a TLS context. In WebAuthn, HTTPS is responsible for validating the origin prior to a registration or authentication ceremony.
The `RP ID`, which is identifying the Relying Party and part of some structures defined in WebAuthn, shown in section 3.3, is directly depending on the origin, because it must be equal to the origin's effective domain, or a registrable domain suffix of the origin's effective domain. The effective domain is either the **domain**, or if that is not given, the **host**. FIDO credentials are bound to the RP ID, to make sure that the credentials are not shared over different origins.[22]

In the TLS case, to provide the effective domain of its origin, the server can use the SAN field, added to the X.509 certificate by the Subject Alternative Name (SAN) extension.[29] The client can then use the Server Name Indication (SNI) extension to specify the host or domain name of the server it intends to communicate with. The authentication of the effective domain provided by the server can then be undertaken directly by TLS itself, by comparing it with the one specified by the client in the SNI.[30] The RP ID is discussed in section 7 of the I-D in appendix A.

Providing and determining the scheme is more difficult, as it is bound to the application protocol. TLS has no inherent knowledge of the specific application that is transporting data over it. Additionally, WebAuthn requires the **scheme** to be set to `HTTPS`.

One possible solution could be, to adapt the `HTTPS` scheme regardless of the actual

application protocol. Although maintaining compatibility with WebAuthn this way, this solution could lead to system components being misled about the actual application protocol being used. It also effectively removes the scheme as a working asset in validating origins, as it would have to be set to `HTTPS` for every single origin anyway.

Another possibility could be, to make the scheme a configurable item for both client and server. The validation of this scheme would fall to the FIDO2 Extension. This would make the use of all possible schemes possible again. How the scheme would be presented by the server and how the validation could be done by the client is still an open question, so the feasibility of this option is not clear. The scheme of the `origin` is discussed in section A.1 of the I-D in appendix A.

5.9. Communicating the ceremony result

In a lot of cases, the Relying Party wants to send the result of a FIDO registration or authentication to the client. This is not done inside the WebAuthn registration or authentication ceremony itself, but happens application specific, after the RP has validated the authenticator response sent by the client. In a typical FIDO context using Webauthn, this result message is sent to the client as an HTTP response by the RP.[2] In the case of the FIDO2 Extension, the server cannot send an additional message with arbitrary data after receiving the *Client Authentication Messages*, so a solution has to be found that does not use the extension mechanic itself. There are different solutions on how the result could be handled.

The easiest solution is to not communicate the result at all, or communicate it only implicitly. This means, that the client can find out the result by interpreting the communication following the ceremonies. If, after finishing a registration ceremony, the client starts an authentication ceremony, it finds out if the registration was successful by either getting an error that the credential is not registered, or the authentication being successful.

After an authentication attempt, the RP will either send the client some form of desired data requiring a successful authentication, or if the RP just closes the connection without sending any data the authentication probably failed. This approach has the problem, that error handling can become difficult on the client side. Especially when a ceremony fails and the client gets no answer at all, the failure must not necessarily have happened during the FIDO ceremony. As most other failure sources have their own corresponding errors, this does not pose a problem if such an error is received. When no error is received, the client

can however never be sure about the exact cause. As retrying a registration or authentication ceremony does not come with a high cost, this should be acceptable in many cases.

If the application does want to explicitly communicate the ceremony result, there are two cases to consider. In the first case the application does not care about this result still being communicated in the TLS handshake. In this case, the result could be sent to the client by the RP simply in the application data using the established TLS connection, or any other method the application desires.

In the other case, the application does want to communicate the result still in the TLS handshake itself. As already said above, this is complicated because the server is not allowed to send an additional arbitrary data after receiving the *Client Authentication Messages*. Jonas Panizza tackled this problem in his thesis, by utilizing the existing TLS alert mechanism.[2] These alerts can be sent by the server even after receiving the authentication message from the client. The alerts are mapped to status codes, which can then be interpreted by the client if the registration/authentication was successful or not.[19] Jonas Panizza used mapping to map the FIDO response to already existing TLS alerts. This can pose the problem, that it is not clear to the client if the original or new meaning of the alert is communicated by the server. However, new TLS alerts can be created, that are extension specific. Extension specific TLS alerts that already exist are for example the `unrecognized_name(112)` alert for the `server_name` extension and `bad_certificate_status_response(113)` alert for the `client_certificate_url` extension.[31] Possible useful new alerts for the FIDO2 TLS 1.3 Extension that would have to be registered with the IANA, are `fido_invalid_credential(122)`, `fido_device_incompatability(123)`, `fido_timeout(124)` and `fido_internal_error(125)`. Even though new alerts are a far better solution than mapping them, the registration of new TLS alerts is very rare and four alerts for a single extension probably not feasible. Instead of four different alerts, a single new alert called `fido2_error(122)` is more realistic and still provides the information to the client, that the error happened because of the FIDO2 Extension. How the alert would be registered with the IANA is described in section 5.10. The corresponding section in the I-D in appendix A is section 12.9.

5.10. Registering the extension and TLS alert with the IANA

For a client and server to communicate using a TLS extension, the extension has to be registered by both sides. Otherwise, the TLS protocol defines, that unknown extensions

have to be ignored. For registering extensions, TLS uses so-called "ExtensionType" values.[19] To make it clear which value means what, they are registered officially with the IANA. The IANA is an organization that is responsible for coordinating key parts of the internet. These include among other things Domain Names, Number Resources and Protocol Assignments.[32] When standardizing a new TLS extension, it also incorporates registering a new ExtensionType value with the IANA. For the FIDO2 TLS 1.3 Extension, the proposed value is 63, as of publishing of this thesis, it is the next free value according to the current registered TLS ExtensionType Values.[31]. The entry also describes, where the extension is allowed to be sent in the TLS handshake. In this case it is in the ClientHello (CH), CertificateRequest (CR) and Certificate (CT).
The new proposed entry into the TLS ExtensionType Values Registry looks as following:

Value	**Extension Name**	**TLS 1.3**	**DTLS-Only**	**Recommended**	**Reference**
63	fido2	CH, CR, CT	N	Y	-

Table 1: Addition to the TLS ExtensionType Values Registry

The new TLS alert defined in section 5.9 would also have to be registered with the IANA. The proposed value is 122, as it is the next free value according to the current registered TLS Alerts.[31]
The new proposed entry into the TLS Alerts Registry looks as following:

Value	**Description**	**DTLS-OK**	**Reference**
122	fido2_error	Y	-

Table 2: Addition to the TLS Alerts Registry

The **Value** and **Reference** entries are not final and would be chosen when the entries are actually registered with the IANA.

5.11. Best use cases of the FIDO2 Extension

The FIDO2 Extension is least complex, when no double handshake has to be used. For authentication, this is possible only for authentication with discoverable credentials, which should make it the desired authentication method for most applications using this extension. For registration, the double handshake cannot be avoided. As registration normally has to be done far less than authentication, this does not necessarily pose a problem. If a double handshake, or the out-of-band distribution of the ticket was to

be avoided completely, the registration could also be done over HTTPS with a classic FIDO2 registration. In scenarios that do not require user interaction, like with Internet of things devices, the registration with the FIDO2 Extension avoids the problem of ticket distribution, as they could easily be pre-configured with valid tickets. Upon first activation, these devices could automatically start a FIDO registration, storing the credentials for example on a platform authenticator.

6. Proof of Concept Implementations

6.1. FIDO2 TLS 1.3 Extension

For this thesis, no completely new Proof of Concept implementation for the extension is created from scratch. Rather, the existing PoC implementation of the FIDO2 TLS 1.3 Extension by Jonas Panizza, presented in his master's thesis "FIDO2 TLS 1.3 Extension: Strong EAP-TLS Authentication for 802.1X Networks", was used as a basis. There are two changes made to the original FIDO2 TLS 1.3 Extension. The other functionality is completely unchanged and therefore not again specified in this thesis. All information regarding the PoC implementation, apart from the changes, can be found in Jonas Panizzas thesis.[2] The changed Proof of Concept implementation can be found on `https://github.com/wede-kind/fidoSSL`.[33]

6.1.1. New Double Handshake

As described in section 5.3, the double handshake, especially the PreHandshake was changed a lot in comparison to the one presented by Jonas Panizza. To prove that the simplified double handshake also works well in practice, the implementation was changed to incorporate the new version. This includes the PreHandshake only including extension data in the `ClientHello` and `CertificateRequest`, as well as sending and encrypting the extension data that was sent in the `Certifacte` in the original implementation in the `ClientHello` of the second handshake instead.

6.1.2. Logging mechanism for the TLS session keys

For the purposes of debugging and analyzation of the sent extension data, it can be useful to decrypt the TLS traffic between the test client and test server of the PoC implementation. To make that possible, the TLS session keys have to be logged by a participating party of the TLS communication. This functionality is added to the FIDO2 TLS 1.3 Extension by using the `SSL_CTX_set_keylog_callback` function of `OpenSSL`.

This generates a key log file when the environment variable `SSLKEYLOGFILE` is set to a valid file location on the side of the client. This key log file can be used for the Wireshark dissector specified in the next section 6.2.

6.2. Wireshark Dissector

To increase trust in the technology, there should be an easy method to see that the extension data of the FIDO2 TLS 1.3 Extension PoC does actually transport the data it promises. For that purpose, Wireshark proves to be a promising application. Wireshark is a tool for capturing network traffic and analyzing the captured packets.[5] When capturing packets including the FIDO2 TLS 1.3 Extension, the extension data cannot simply be observed. There are two reasons for that, that have to be addressed.

6.2.1. TLS transport protection

The extension data sent in the `CertificateRequest` and the `Certificate` cannot be observed, as they are encrypted by the TLS transport protection. Wireshark offers the ability to decrypt this data, by providing a `(Pre)-Master-Secret log file`. This log file can be provided with the help of the `SSLKEYLOGFILE` mechanism described in section 6.1.

6.2.2. Making the extension data human-readable

The extension data of the FIDO2 TLS 1.3 Extension is encoded in one CBOR blob per packet. This CBOR blob is not human-readable. Wireshark has so-called dissectors, that are used for decoding parts of the protocol of the viewed packet.[34] Wireshark offers the possibility to add own dissectors. There are many choices one has to take when writing a Wireshark dissector, which are explained following in respect to the FIDO2 TLS 1.3 Extension:

- The dissectors can be either built-in to Wireshark, or written as a plugin. The plugin variant was chosen for this thesis, as it offers all functionality needed in the case of the FIDO2 Extension and makes development and distribution of the dissector easier.
- Plugins can be written in either C or Lua. As writing a dissector in Lua is seemingly simpler and the integration of Lua plugins into Wireshark is very easy, it was chosen over C.

- In Wireshark there are two types of dissectors. Post-dissectors and chained dissectors. Post-dissectors run after all other dissectors, which has the advantage, that all protocol fields are already set and can be easily worked with. The biggest disadvantage is, that they have to run against every packet. Chained dissectors run, after another dissector for a specific part of a packet finished. In this case, it would run after the TLS dissector is done. This has the advantage, that they only run for relevant packets. A big disadvantage in the case of the FIDO2 extension is, that chained dissectors have to be port specific, or they get very difficult to handle. As TLS is not port specific, the better choice in this case is a post-dissector, which can easily handle arbitrary ports.

The resulting Wireshark plugin presents the extension data of the FIDO2 TLS 1.3 Extension in an easily understandable and human-readable form. It works out of the box and does not need any configuration. The plugin can be found on `https://github.com/wede-kind/fido-tls-dissector`.[35] A guide for installation and usage of the plugin, as well as examples of dissected packets, can be found in appendix B.

7. Conclusion and Future Work

FIDO2 is a proved method for passwordless authentication. As shown in this thesis, integrating FIDO2 into the TLS handshake in the form of an extension can help to decouple FIDO2 from the web environment and HTTPS it is bound to with traditional WebAuthn. This makes passwordless authentication accessible to far more applications.

Methods developed in previous groundwork could be enhanced. This includes a simplification of the double handshake, an improvement of the encryption by obscuring data length with padding and better distribution of the key used for encryption that allows the server to hold no client specific states between handshakes. New ideas were developed for validation of the origin as well as for the communication of the ceremony result.

The groundwork for possible future standardization has been laid with the Internet-Draft “FIDO2 TLS 1.3 Extension”. It includes a complete overview over all parts of the FIDO2 Extension and clear instructions concerning protocol flow and message specifications. It should help as a guideline for future implementations and make sure, that all future implementation based on the I-D are completely compatible with each other. The Internet-Draft also decouples the FIDO2 Extension from specific libraries like OpenSSL, which were used in previous research.

Some questions regarding the extension and the I-D are still open, which could be tackled in future work:

7.1. Additional Authentication with AES-GCM

As described in section 5.4, `AEAD_AES_256_GCM` is used to encrypt and authenticate the sensitive client data in the ClientHello and CertificateRequest of the registration and authentication with server-side credentials. `AEAD_AES_256_GCM` also allows adding associated data to the encryption, which is not encrypted itself, but still authenticated by being part of the GCM tag.[27] In the case of the `ClientHello` in a double handshake, the `ephemeral user ID` is currently not encrypted and therefore also not authenticated. By adding the `ephemeral user ID` to the associated data, it would be authenticated, which could provide useful when being processed by the server.

7.2. Determining the Origin's scheme

In section 5.8, the concepts of the origin and PR ID are discussed. While a satisfactory solution for the `host` of the origin and therefore the RP ID, was found, there exists no satisfactory solution for the `scheme` yet. Both using **HTTPS** regardless of application protocol and configuration of the scheme on client and server side can have unforeseeable consequences that have to be further discussed and tested.

7.3. Development of an easily recognizable Dummy Certificate

As described in 5.1, the extension ensures that the client has access to a client certificate and therefore the extension mechanic in the *Client Authentication Messages*, by proposing the usage of a dummy certificate. To make sure the server does not confuse the dummy certificate with a real certificate and can safely associate it with the FIDO2 Extension, a FIDO specific dummy certificate could be designed.

7.4. Further Development of the Internet-Draft

While the Internet-Draft, shown in Appendix A does provide a solid basis for potential future standardization of a FIDO2 Extension, it lacks discussion in working groups. Normally, I-Ds go through multiple loops of discussion and enhancing before finally being published as a RFC. To kickstart discussion and advance potential standardization, the I-D is planned to be presented to the FIDO Alliance.

7.5. Proof of Concept Implementation based on the I-D

While the POC implementation described in 6.1 shows that the concepts of the FIDO2 Extension work in practice, it does not conform to the Internet-Draft. This includes the protocol flow, as well as the message specifications. A future POC that conforms 100% with the I-D could help further strengthening the trust in the concept, while also uncovering potential flaws in the current design. The Wireshark dissector would then also have to be updated to conform with the new implementation.

References

[1] FIDO ALliance. Fido alliance, 2026. https://fidoalliance.org/ [Accessed: 22.01.2026].

[2] Jonas Panizza. Fido2 tls 1.3 extension: Strong eap-tls authentication for 802.1x networks, 2025.

[3] Allison J. Mankin and Stephen Hayes. Requirements for IETF Technical Publication Service. RFC 4714, October 2006.

[4] Rfc editor. https://www.rfc-editor.org/ [Accessed: 20.02.2026].

[5] Wireshark homepage. https://wiki.wireshark.org/ [Accessed: 20.02.2026].

[6] Git repository: I-d-fido2-tls-1-3-extension. https://github.com/wede-kind/I-D-FIDO2-TLS-1-3-Extension/commit/45ddb44f25bc3f0fa7b6d087f86644edd67cba63.

[7] Russ Housley and Leslie Daigle. The RFC Series and RFC Editor. RFC 8729, February 2020.

[8] Aaron Falk. Definition of an Internet Research Task Force (IRTF) Document Stream. RFC 5743, December 2009.

[9] Nevil Brownlee and Bob Hinden. Independent Submission Editor Model. RFC 8730, February 2020.

[10] Peter Saint-Andre. RFC Editor Model (Version 3). RFC 9280, June 2022.

[11] IETF. Internet-draft author resources, 2026. https://authors.ietf.org/en/home [Accessed: 22.01.2026].

[12] Git repository: xml2rfc. https://github.com/ietf-tools/xml2rfc/commit/c4734981da3758af34a8806c863080ef7eefdb39.

[13] Sandy Ginoza and Heather Flanagan. RFC Style Guide. RFC 7322, September 2014.

[14] Eric Rescorla and Brian Korver. Guidelines for Writing RFC Text on Security Considerations. RFC 3552, July 2003.

[15] Michelle Cotton, Barry Leiba, and Dr. Thomas Narten. Guidelines for Writing an IANA Considerations Section in RFCs. RFC 8126, June 2017.

[16] Iesg statement: Normative and informative references. Technical report, IESG, 2006.

[17] Nevil Brownlee and IAB. SVG Drawings for RFCs: SVG 1.2 RFC. RFC 7996, December 2016.

[18] IETF. Ietf datatracker's submission tool, 2026. https://datatracker.ietf.org/submit/ [Accessed: 22.01.2026].

[19] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[20] IANA. Transport layer security (tls) extensions - version from 24 december 2025. https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml.

[21] FIDO Alliance. Client to authenticator protocol (ctap), review draft, march 21, 2023, 2023.

[22] World Wide Web Consortium. Web authentication: An api for accessing public key credentials level 2, 2021.

[23] Jan-Frederik Rieckers and Stefan Winter. EAP-FIDO. Internet-Draft draft-ietf-emu-eap-fido-01, Internet Engineering Task Force, March 2025. Work in Progress.

[24] Tom-Lukas Johann Breitkopf. Fido2 als tls-1.3-erweiterung, 2020.

[25] Mario Freund. Fido2-erweiterung von tls 1.3 in einer gängigen kryptobibliothek, 2021.

[26] Stefan Santesson. TLS Handshake Message for Supplemental Data. RFC 4680, October 2006.

[27] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008.

[28] Web Hypertext Application Technology Working Group. Html living standard - version from 23 january 2026, January 2026.

[29] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[30] Donald E. Eastlake 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, January 2011.

[31] IANA. Transport layer security (tls) parameters - version from 13 february 2026. https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-6.

[32] Joseph A. Salowey and Sean Turner. IANA Registry Updates for TLS and DTLS. RFC 8447, August 2018.

[33] David Wedekind. Git repository: fidossl. https://github.com/wede-kind/fidoSSL/commit/80b206f7d636dbcb9f53b0ab57ad5068af02fc10.

[34] Wireshark packet dissection documentation. https://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html [Accessed: 20.02.2026].

[35] David Wedekind. Git repository: fido-tls-dissector. https://github.com/wede-kind/fido-tls-dissector/commit/f6bce093e0ce48ab7a36fa75f47b83506f55758f.

[36] Git repository: Lua-cbor. https://github.com/Kristopher38/lua-cbor/commit/634b929b6e2773b3908912a37006c95852765eac.

8. Appendix

A. Internet-Draft "FIDO2 TLS 1.3 Extension"

This appendix includes the complete PDF version of the Internet-Draft "FIDO2 TLS 1.3 Extension".

Workgroup:	Internet Engineering Task Force
Internet-Draft:	draft-fido2-tls-1.3-extension-00
Published:	23 February 2026
Intended Status:	Informational
Expires:	27 August 2026
Author:	D. Wedekind, Ed. *Humboldt-Universität zu Berlin*

FIDO2 TLS 1.3 Extension

Abstract

FIDO2 enables passwordless authentication and uses passkeys instead. These passkeys omit many of the problems traditional passwords have, for example phishing, and credential reuse. As FIDO2 was developed for the web context, it is currently bound to HTTPS. This document describes a mechanism that allows FIDO2 to be used in TLS 1.3 via a TLS 1.3 extension. This extension aims to decouple FIDO2 from HTTP to make FIDO2 easily accessible for all applications built on TLS 1.3.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 27 August 2026.

Copyright Notice

Table of Contents

1. Introduction

Passwords are still often used in account authentication despite their many problems like phishing, and credential reuse. Passwordless alternatives for authentication are getting more common. The FIDO Alliance is an open industry association, that develops standards for passwordless authentication. FIDO2 is a project of the FIDO Alliance, that allows easy passwordless authentication over the web. FIDO2 consists of the W3C Web Authentication (WebAuthn) standard [WebAuthn], and the FIDO Client to Authenticator Protocol 2 (CTAP2) [FIDO-CTAP2]. CTAP2 allows for communication between an external, and portable authenticator such as a hardware security key or a smartphone, and a website or account using the authenticator. The W3C Web Authentication standard defines an API enabling the creation, and use of strong, attested, scoped, public key-based credentials by web applications. The standard is designed for use with HTTPS.

This documents goal is to describe a solution to decouple FIDO2 from HTTPS by making registration, and authentication possible already in the TLS handshake. This would enable all applications that use TLS 1.3 to use FIDO2 for registration, and authentication of FIDO credentials.

This document specifies a new TLS extension, "FIDO2", that applications can use for passwordless authentication via the TLS handshake. This extension is only applicable for TLS 1.3 [RFC8446].

The extension is based on WebAuthn, and is mostly scoped on the communication between the client, and server in a registration or authentication ceremony. The communication between the client, and authenticator, as well as the processing of the data by the client, and server, is semantically unchanged except of the data encoding, and happens as is specified in WebAuthn, unless specifically mentioned, and changed.

2. Conventions and Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses terminology from CTAP and WebAuthn. The FIDO specific terminology is defined in [FIDO-Glossary].

The terms "client" and "server" used in this document have implications regarding TLS and Webauthn and are defined as follows:

client: Consolidates the functionality of the WebAuthn Client, and the Relying Party application defined in [WebAuthn]. Additionally manages the TLS connection as a client in [RFC8446].

server: Incorporates the functionality of a WebAuthn Relying Party defined in [WebAuthn], and manages the TLS connection as a server in [RFC8446].

3. Data Structures

The data structures defined in this document are mainly CBOR data objects, which are described with the Concise Data Definition Language (CDDL) laid out in [RFC8610].

The FIDO2 TLS 1.3 Extension also uses enumerations, which are difficult to express in CDDL. These are instead defined and encoded according to the conventions laid out in Section 3.5 of [RFC8446]. The enums are mapped to CBOR uints, described in comments in the CBOR data objects.

4. Design Overview

The FIDO2 Extension can be used when a registration or authentication of a FIDO credential using the TLS 1.3 handshake is desired. It makes use of the TLS handshake extension mechanic to enable registration or authentication of FIDO credentials in exclusively the TLS handshake messages. TLS 1.3 is very suitable as a basis for FIDO, as it allows easy authentication of the server's identity, which is very important for FIDO to uphold phishing resistance. The basis for the design of the extension is the specification "Web Authentication: An API for accessing Public Key Credentials Level 2" (WebAuthn) [WebAuthn]. WebAuthn defines an API that is intended for the web and HTTPS. The extension makes FIDO2 independent of specific application protocols like HTTPS. It is mostly scoped to the transportation of data between the client and the Relying Party. The processing of the data by the client, and server, as well as the communication between the client, and the authenticator still happens as defined in [WebAuthn], apart from a few changes that are necessary, because of challenges not present for WebAuthn.

4.1. Challenges

Specifying the FIDO2 Extension outside the web and HTTPS environment produces many challenges and requires changes to both the protocol flow and message specifications defined in WebAuthn. These challenges are:

- Transmission of arbitrary data in the TLS handshake is limited to client->server->client. Relevant in Section 8, and Section 11.5.
- The exchange of data is not happening in an encrypted and authenticated environment for a part of a TLS 1.3 handshake. Discussed in Section 8.
- WebAuthn uses JSON objects for communication, which is not very space efficient. TLS extension data is limited to 2^16-1 bytes. Additionally TLS records are size constrained to 2^14 bytes and will be fragmented if the TLS message is larger, which should be avoided whenever possible for efficiency and robustness reasons. Discussed in Section 12.
- In the original FIDO2 registration, the outer web context is used to conduct which users are permitted to register a new FIDO credential with the Relying Party. The registration ceremony itself lacks such a control mechanism. Explained in Section 9.
- WebAuthn uses the concept of an origin, defined in the [HTML-Living-Standard]. It is used by the client to authenticate the server identity. It has two mandatory components. The first is the host, which is also used to validate a so called Relying Party Identifier (RP ID), which is an important part of credentials. The determination of the host and RP ID is described in Section 7. The second is a scheme, which is bound to the application protocol. As TLS is a transport protocol, which is independent of the overlying application protocol, the concept of a scheme does not inherently apply. Additionally, in the WebAuthn specification, the scheme must be "https". The scheme is discussed in Appendix A.1.

5. FIDO2 Extension

This document defines the following "fido2" extension:

```
enum {
    fido2(63), (65535)
} ExtensionType;
```

Clients **MAY** send this extension in the ClientHello and in a Certificate. Servers **MAY** send this extension in the CertificateRequest. The structures that are contained in the extension are defined in Section 12. How and when this extension and the corresponding structures are sent is explained in Section 11.

6. Message Flow Overview

The FIDO2 Extension differentiates three main message flows. Registration, equivalent to the message flow in the WebAuthn Registration Ceremony, Authentication with non-discoverable credentials, and Authentication with discoverable credentials, which are equivalent to two variants of the WebAuthn Authentication Ceremony. These message flows are different concerning the exchanged data, and processing by the client, and server, but follow the same pattern in the TLS handshake. The precise message flows are defined in Section 11.

Figure 1 shows two consecutive simplified TLS 1.3 handshakes between a client and a server, incorporating generalized FIDO2 extension messages. The registration and authentication itself happens in the second handshake. In the first handshake, or "PreHandshake", a key is exchanged for encryption of sensitive client information in the second handshake, further explained in Section 8. As shown in the figure, the "fido2" extension is sent in the ClientHello, CertificateRequest and client Certificate.

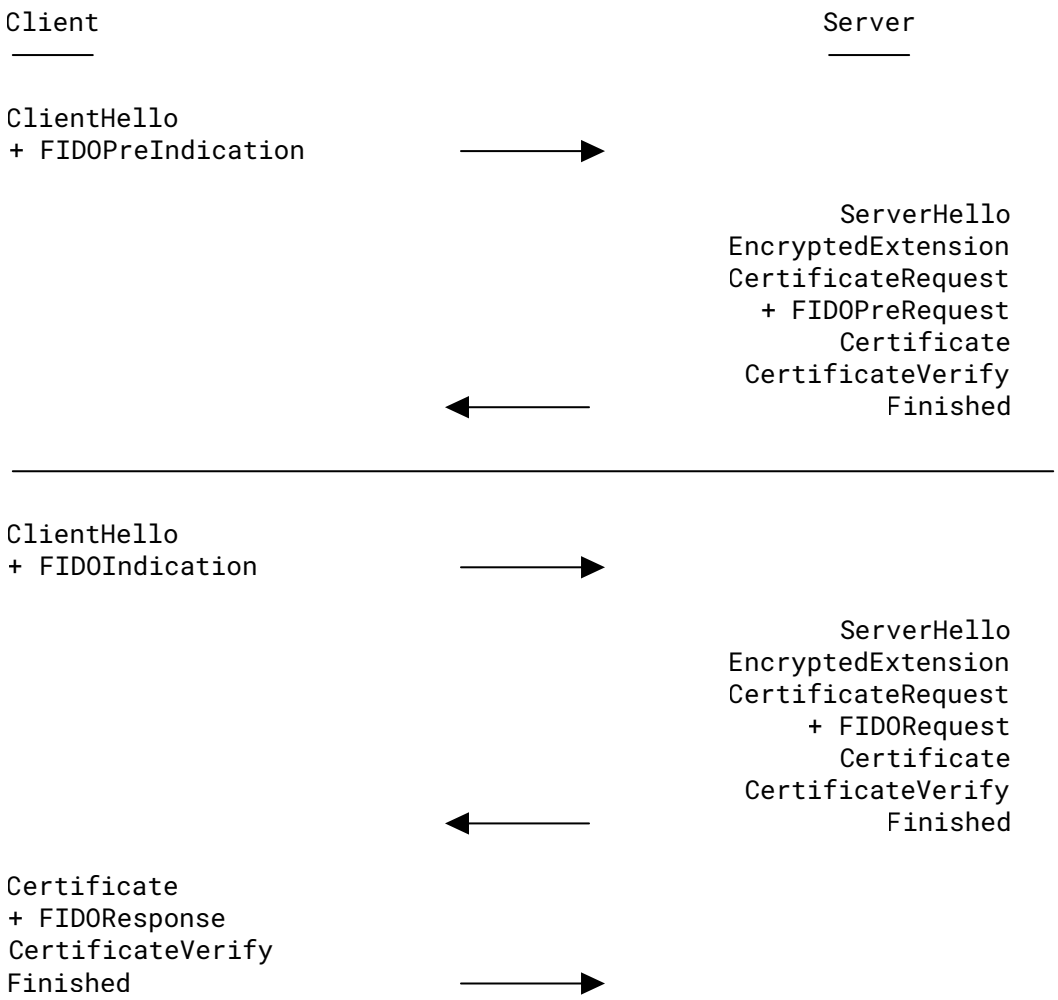

Figure 1: Message Flow of the "fido2" extension generalized for both registration and authentication

7. Determining the RP ID

The FIDO specification uses a Relying Party ID and origin as a method to ensure the client is communicating with a trusted Relying Party. It is also used to make sure a public key credential can only be used to authenticate a user with the same Relying Party it was registered for. This avoids phishing attacks. The origin consists of a scheme, host, and optionally a port, and domain, as defined in the HTML Living Standard [HTML-living-standard-origin]. According to WebAuthn, the RP ID is based only on the origin's domain name and does not include a scheme or port. This means it is only dependent on the origin's host, or domain, further called effective domain. The scheme of the origin is discussed in Appendix A.1.

To determine the effective domain of the origin, the FIDO2 Extension relies on the servers X.509 certificate, which **MUST** be provided by the server. The X.509 certificate **MUST** include the subject alternative name (SAN) extension FIDO2 Extension defined in [RFC5280], in which it provides its host in the SAN field. The client **MUST** use the Server Name Indication (SNI) extension, defined in [RFC6066], to indicate the effective domain of the server it intends to communicate with. The authentication of the server host is then done by TLS by comparing the effective domain name in the SNI field with the effective domain provided in the SAN field of the server certificate.

The server is allowed to provide an RP ID different to the origin's effective domain provided in the SAN field in its Registration and Authentication Request messages defined in Section 12. In this case, the client **MUST** validate, that the RP ID is equal to a registrable domain suffix of the origin's effective domain.

8. Encryption

The WebAuthn ceremonies rely on a secure communication channel, so no additional encryption of any value exchanged in the ceremonies is necessary. The FIDO2 Extension can only rely on the protection already offered in the TLS handshake. In the case of Registration, and Authentication with non-discoverable credentials, sensitive data has to be sent in the "fido2" extension sent in the ClientHello message, and in the CertificateRequest. Extensions in the ClientHello message are not encrypted. Extensions in the CertificateRequest are encrypted, but vulnerable to a man-in-the-middle attack in the case of the FIDO2 Extension. An attacker could intercept and replay the FIDOIndication message of either the registration, or authentication, to the server. The server then sends back the Request message of the registration, or authentication, which includes informations about the client the Indication message was intercepted from. For these reasons, all sensitive client data has to be encrypted in these messages.

The encryption method used is AEAD_AES_256_GCM, described in [RFC5116], further also called AES-256-GCM. It is part of the "TLS_AES_256_GCM_SHA384" cipher suite, which is a mandatory-to-implement cipher suite for TLS 1.3 defined in [RFC8446], Section 9.1. This makes sure that every possible client has access to the encryption method. Additionally to encryption, AES-256-GCM also offers authentication of the encrypted data. To simplify the encryption process, if a message includes multiple values that have to be encrypted, they are combined to one encrypted_data object.

AES-256-GCM uses 256-bit encryption keys. The PreHandshake and second handshake are connected by an ID called the "ephemeral user ID". The key and ephemeral user ID are distributed from the server to the client in the PreHandshake, defined in Section 11.2. The server has two different options on how it wants to generate the 256-bit key and ephemeral user ID:

- The server generates a random key and random ephemeral user ID for every PreHandshake, and saves both together in a database. The key can then be looked up when an ephemeral user ID is received. The server **MUST** ensure, that no key is purposely reused. As there is no guarantee the client ever initiates a second handshake, the ephemeral user ID, and key **SHOULD** be deleted by the server after a reasonable time span of a few minutes at most.

- The server generates only a random ephemeral user ID for every PreHandshake. The key is then derived from the user ID by using a master secret only the server knows. A possibility is to derive the 256-bit key as following:

```
256_bit_key = SHA256(ephemeral_user_id||master_secret)
```

  As this method provides no forward secrecy, a new master secret **MUST** be generated every few minutes, with the old master secret being permanently deleted. As the decryption in the second handshake could fail, because the master secret changed right after the PreHandshake, the server **MAY** hold the master secret that is one iteration old, which can be tried if decryption with the newest master secret fails.

  This option has the advantage, that the server does not have to keep client specific information between the two handshakes. It is especially useful for servers that have multiple physical servers a client can connect to. When load balancing is used, the new TCP connection opened by the client for the second handshake does not necessarily happen with the same physical server. With this option, only the master secret has to be synchronized between the physical server.

8.1. Padding

The length of the input can be directly derived from an AES-256-GCM output. This potentially enables third parties to identify users by the length of the user name and user display name. To counteract this attack vector, the user name, and user display name **MUST** be padded to their maximum length of 256 bytes before encryption. The padding is one '1' bit followed by as many '0' bits as necessary to reach the necessary length.

9. Ticket for Registration Control

In the original FIDO2 registration ceremony, the Relying Party may use additional authentication steps to control if a user is permitted to register a new credential. This interaction mostly happens over a website in a browser and includes authentication methods like passwords, or email. For the FIDO2 Extension, which can not rely on a context like a browser, a ticket mechanic is instead implemented.

The "ticket" is a pre-shared secret only known by the server and a valid client. It is distributed out-of-band. The channel over which the ticket is distributed is freely chosen, but **MUST** ensure, that the ticket is never visible to untrusted third parties. The ticket **MUST** be 256 bytes in size. As the ticket **SHOULD** be primarily used as a control mechanic to ensure that a client is to allow a new credential with the server, it can be completely randomly generated. The ticket **MAY** include additional semantic, that can help the server when validating the ticket, but it **MUST** have a random component of 128 bits, that makes sure that a server never distributes an exact same ticket twice in a reasonable time span.

The ticket is sent to the server in the Registration Request and **MUST** be encrypted by the client, as it is not encrypted by TLS in the ClientHello.

10. Dummy Certificate

The Certificate is the only possibility for the client to send an extension to the server in the client authentication messages. The FIDO2 Extension only depends on the client certificate for transporting the extension, but the client certificate does not have to actually be a valid certificate for the specific client. To make sure, that the client has access to a certificate, a "dummy certificate" **MAY** be used instead. This certificate **MUST** seem valid in order for the TLS handshake to continue, but does not actually authenticate the client. If a dummy certificate is used, it **MUST** be obvious to the server that the certificate is only used to transport the "fido2" extension.

11. Protocol flow

The protocol flow of the FIDO2 Extension differentiates registration of a new credential, authentication of a discoverable credential, and authentication of a server-side credential. The following sections describe, when and how a client or server **MAY** send a "fido2" extension. In general, a client or server **MUST NOT** respond with a "fido2" extension, if it did not receive a "fido2" extension in the prior message. When a TLS version negotiated is less than 1.3, the server **MUST** ignore this extension. If a client, or server receives a "fido2" extension with a CBOR structure including a different message_type than fitting the protocol flow specified in the following sections, it **MUST** abort the connection with an "illegal_parameter" alert. The processing of the data in the registration and authentication ceremonies of the client and Relying Party is defined as in [WebAuthn] and not changed by the FIDO2 TLS 1.3 extension, unless otherwise stated in specific cases. The processing of the data is not specifically explained in the following sections, but happens implicitly before a new CBOR object is sent to the communication partner. If an error occurs in the processing of data as defined in [WebAuthn], the connection **MAY** be aborted with a "fido2_error" alert by the client or server respectively, depending on the severity of the error. An abortion of the connection caused by an error in the processing of data is only specified, if the processing is changed by the FIDO2 TLS 1.3 extension. Every mention of encryption implicitly refers to section Section 8.

11.1. Error Communication

This document allocates a new alert value in the TLS Alert Registry [RFC8446]:

```
enum {
  /* ... */
  fido2_error(122),
  /* ... */
  (255)
} AlertDescription;

This alert is always fatal.
```

11.2. PreHandshake

The PreHandshake establishes the 256-bit key used for encryption in the second handshake. It consists of a PreIndication sent from the client to the server and a PreResponse

11.2.1. PreIndication

A client that intends to either register a new credential, or authenticate a server-side credential with the Relying Party **SHALL** send a "fido2" extension in its ClientHello. The body of the sent extension consists of a PreIndication CBOR structure defined in Section 12.1.

11.2.2. PreResponse

If the server receives a ClientHello with a "fido2" extension containing a PreIndication with a message_type of 1, the server **MAY** send back a "fido2" extension in its CertificateRequest. The body of the sent extension consists of a PreResponse CBOR structure defined in Section 12.2.

11.3. Registration Handshake

In the registration handshake, the messages necessary to perform a successful registration ceremony are exchanged. A registration handshake **MUST** be preceded by a PreHandshake, as certain data in the relevant CBOR structures is encrypted and authenticated with the 256-bit key established in the PreHandshake.

11.3.1. Registration Indication

If the client receives a CertificateRequest with a "fido2" extension containing a PreReponse with a message_type of 2 and wants to continue with the registration of a new credential, the client **MAY** open a new, or reuse the existing TCP connection with the server and send a fresh ClientHello including a "fido2" extension. The body of the sent extension consists of a RegistrationIndication CBOR structure defined in Section 12.3. A reason why the client would not continue is, if it deems the 256-bit key received in the PreResponse unfit. In this case, the client **MUST** abort the connection of the PreHandshake with a "fido2_error" alert.

11.3.2. Registration Request

If the server receives a ClientHello with a "fido2" extension containing a RegistrationIndication with a message_type of 3, it uses the "ephemeral_user_id" to look up the 256-bit key associated with the handshake. It then uses the 256-bit key to authenticate, and decrypt all encrypted values. If the ticket is a valid ticket for creating a new credential, according to the server, it **MAY** answer with a "fido2" extension in its CertificateRequest. The body of the sent extension consists of a RegistrationRequest CBOR structure defined in Section 12.4. If the server does not find an entry for the "ephemeral_user_id", can not successfully decrypt the values, or does not deem the ticket to be valid, it **MUST** abort the connection with a "fido2_error" alert.

11.3.3. Registration Response

If the client receives a CertificateRequest with a "fido2" extension containing a RegistrationRequest with a message_type of 4, the client uses the 256-bit key received in the PreResponse to decrypt all encrypted values. It **MAY** then answer with a "fido2" extension in its Certificate. The body of the sent extension consists of a RegistrationResponse CBOR structure defined in Section 12.5. If the authentication of the decrypted data fails, the client **MUST** abort the connection with a "fido2_error" alert. To make sure the client has access to a valid certificate in which it can send the extension data, the client **MAY** use a dummy certificate as specified in Section 10.

11.4. Authentication Handshake

In the authentication handshake, the messages to perform an authentication of a registered credential are exchanged. The FIDO2 TLS 1.3 differentiates two protocol flows for authentication with discoverable credentials, and authentication with server-side credentials. The reason is, that no PreHandshake is necessary for authentication with discoverable credentials, while authentication with server-side credentials **MUST** be preceded by a PreHandshake. Some data in the authentication with server-side credentials is encrypted and authenticated with the 256-bit key received in the PreHandshake. The two authentication variants use different CBOR structures for the indication, while sharing the structures for request and response.

11.4.1. Authentication Indication (discoverable credential)

A client that intends to authenticate a discoverable credential with the Relying Party **SHALL** send a "fido2" extension in its ClientHello. The body of the sent extension consists of an AuthenticationIndicationDiscoverable CBOR structure defined in Section 12.6. The authentication with discoverable credentials does not need a prior PreHandshake, because no client specific data has to be sent in the unencrypted ClientHello.

11.4.2. Authentication Indication (server-side credential)

If the client receives a CertificateRequest with a "fido2" extension containing a PreResponse with a message_type of 2 and wants to continue with authentication of a server-side credential, the client **MAY** open a new, or reuse the existing TCP connection with the server and send a fresh ClientHello including a "fido2" extension. The body of the sent extension consists of a AuthenticationIndicationServerSide CBOR structure defined in Section 12.7. A reason why the client would not continue is, if it deems the 256-bit key received in the PreResponse unfit. In this case, the client **MUST** abort the connection of the PreHandshake with a "fido2_error" alert.

11.4.3. Authentication Request

If the server receives a ClientHello with a "fido2" extension containing an AuthenticationIndicationDiscoverable with a message_type of 6, or AuthenticationIndicationServerSide CBOR object with a message_type of 7, it acts depending on the received CBOR object:

- AuthenticationIndicationDiscoverable:

The server **MAY** answer with a "fido2" extension in its CertificateRequest. The body of the sent extension consists of an AuthenticationResponse CBOR structure defined in Section 12.8, that **MUST NOT** include the optional value allow_credentials.

- AuthenticationIndicationServerSide:

  The server uses the "ephemeral_user_id" received in the AuthenticationIndicationServerSide object to look up the 256-bit key associated with the handshake. It then uses the 256-bit key to authenticate, and decrypt all encrypted values. It **MAY** then answer with a "fido2" extension in its CertificateRequest. The body of the sent extension consists of an AuthenticationResponse CBOR structure defined in Section 12.8, that **MUST** include the optional value allow_credentials. If the authentication of the decrypted data fails, the server **MUST** abort the connection with a "fido2_error" alert.

11.4.4. Authentication Response

If the client receives a CertificateRequest with a "fido2" extension containing an AuthenticationRequest CBOR object with a message_type of 8, it acts depending on if it initiated the Handshake for authentication with discoverable credentials or authentication with server-side credentials and its associated CBOR objects:

- Authentication with discoverable credentials:

  The client **MAY** answer with a "fido2" extension in its Certificate. The body of the sent extension consists of a AuthenticationResponse CBOR structure defined in Section 12.8, that **MUST** include the optional value user_handle.

- Authentication with server-side credentials:

  The client uses the 256-bit key received in the PreResponse to decrypt all encrypted values. It **MAY** then answer with a "fido2" extension in its Certificate. The body of the sent extension consists of an AuthenticationResponse CBOR structure defined in Section 12.8. If the decryption of the values fails, the client **MUST** abort the connection with a "fido2_error" alert.

To make sure the client has access to a valid certificate in which it can send the extension data, the client **MAY** use a dummy certificate as specified in Section 10.

11.5. Result Communication

Communicating the result of a registration or authentication ceremony in the TLS handshake is optional, as the result communication is also not part of the ceremony itself in WebAuthn, but rather application specific. In the FIDO2 Extension, a server **MAY** indicate, that a registration or authentication failed, by sending a "fido2_error" alert to the client, after validating the RegistrationRequest, or AuthenticationRequest respectively.

12. Message Specifications

Specifies the structures that can be sent in a "fido2" extension. The FIDO2 Extension uses Concise Binary Object Representation (CBOR) for all structures, as it is more space efficient than JSON mostly used in WebAuthn, but still offers the same flexibility. CBOR is already utilized by CTAP

for communication between the client and the authenticator, so it introduces no new dependency for the client, and additionally decreases complexity because it removes recoding between JSON and CBOR. The enumerations are only used for visualization purposes and not part of the final CBOR structures. They are mapped to uints instead. Every value that is encrypted or padded according to its description follows the mechanisms specified in Section 8.

12.1. PreIndication Message

```
PreIndication = [
    message_type: 1,
]
```

message_type: An identifier used for easier message processing and debugging. In the ClientHello message it also serves the RP to identify the client's desired action.

12.2. PreResponse Message

```
PreResponse = [
    message_type: 2,
    ephemeral_user_id: bstr . size 256,
    256_bit_key: bstr .size 32
]
```

message_type: An identifier used for easier message processing and debugging.

ephemeral_user_id: An ephemeral reference that is used to link the PreHandshake and the second handshake. It is randomly generated by the RP.

256_bit_key: A key used for symmetric encryption with AES-256-GCM in the second handshake.

12.3. Registration Indication Message

```
RegistrationIndication = [
    message_type: 3,
    ephemeral_user_id: bstr .size 256,
    encrypted_data
]
```

message_type: An identifier used for easier message processing and debugging. In the ClientHello message it also serves the RP to identify the client's desired action.

ephemeral_user_id: An ephemeral reference that is used to link the PreHandshake, and the second handshake. Received by the client from the RP in the PreHandshake.

encrypted_data: CBOR array including sensitive data, that gets padded, and encrypted as described in Section 8.

```
encrypted_data = [
    user_display_name: bstr .size (1..256),
    ticket: bstr .size 256
]
```

user_display_name: The user display name is a human-palatable name for the user account. Intended only for display purposes. Chosen by the user. The user display name is defined in [WebAuthn], Section 5.4.3.

ticket: The ticket is explained in Section 9.

12.4. Registration Request Message

```
RegistrationRequest = [
    message_type: 4,
    rp_id: tstr .size (1..256),
    rp_name: tstr .size (1..256),
    challenge: bstr .size (16..64),
    pub_key_cred_params = [1* pub_key_cred_param],
    encrypted_data,
    ? optionals
]
```

message type: An identifier used for easier message processing and debugging.

rp_id: Identifier of the Relying Party. The RP ID is discussed in Section 7.

rp_name: Human-palatable identifier of the Relying Party. Intended only for display purposes. Defined in [WebAuthn], Section 5.4.1.

challenge: Challenge that MUST be randomly generated by the Relying Party in an environment they trust. Intended to be used for generating the newly created credential's attestation object. Defined in [WebAuthn], Section 13.4.3.

pub_key_cred_params: List of the desired properties of the credential to be created. Ordered from most preferred to least preferred. Defined in [WebAuthn], Section 5.4.

encrypted_data: CBOR array including sensitive data, that gets padded, and encrypted as described in Section 8.

optionals: Optional parameters. Defined in [WebAuthn], Section 5.4.

```
pub_key_cred_param = [
    type: uint .size 1, ; mapped to enum of PublicKeyCredentialType
    alg: uint .size 3   ; mapped to enum of CoseAlgorithmIdentifier
]
```

```
enum {
    public-key(1)
} PublicKeyCredentialType
```

```
/* selection */
/* https://www.iana.org/assignments/cose/cose.xhtml#algorithms */
enum {
    COSE_ES256(-7),
    COSE_ES384(-35),
    COSE_ES512(-36),
    COSE_EDDSA(-8),
/* https://github.com/Yubico/libfido2/blob/main/src/es256.c#L90 */
    COSE_ECDH_ES256(-25),
    COSE_RS256(-257),
    COSE_RS1(-65535)
} CoseAlgorithmIdentifier
```

```
encrypted_data = [
    user_name: bstr .size (1..256),
    user_display_name: bstr .size (1..256),
    user_id: bstr .size 64,
    ? [* public_key_credential_descriptor] ; exclude_credentials
]
```

user_name: The user name is a human-palatable identifier for the user account. Chosen by the RP. **MAY** be the same as the user display name received in the RegistrationIndication. **MUST** be unique among all user names registered for clients. The user name is defined in [WebAuthn], Section 5.4.1.

user_display_name: The user display name is a human-palatable name for the user account intended only for display purposes. Mirrors the encrypted user display name received in the RegistrationIndication. The user display name is defined in [WebAuthn], Section 5.4.3.

user_id: A user ID is a user handle of the user account entity. The user ID is defined in [WebAuthn], Section 5.4.3.

exclude_credentials Optional parameter. List of public key credentials. The client is requested to return an error if the new credential would be created on an authenticator that contains a credential on that list. Used by the RP to limit creation of multiple credentials for the same account on the same authenticator. Defined in [WebAuthn], Section 5.4.

```
public_key_credential_descriptor = [
    type: uint .size 1,            ; mapped to enum of
                                     PublicKeyCredentialType
    id: bstr .size (16..256),
    ? transports: [* uint .size 1] ; mapped to enum of
                                     AuthenticatorTransport
]
```

type: Describes valid credential types.

id: Contains the credential ID of the public key credential.

transports: Optional parameter. List of hints as to how clients might communicate with a particular authenticator. Represents the RPs best belief as to how an authenticator might be reached.

```
optionals = {
    ? 1 => uint .size 4                    ; timeout
    ? 2 => authenticator_selection_criteria ;
                                     authenticator_selection_criteria
    ? 4 => uint .size 1 .default 1         ; attestation,
                                     mapped to enum of
                                     AttestationConveyancePreference
    ? 5 => extensions                      ; extensions
}
```

timeout: Time in Milliseconds that the caller is willing to wait for the call to complete. MAY be overridden by the client.

authenticator_selection_criteria: Used if RP wishes to select appropriate authenticators for the credential creation. Defined in [WebAuthn], Section 5.4.4.

attestation Used if RP wishes to express their preference for attestation conveyance. Attestation conveyance is defined in [WebAuthn], Section 5.4.7.

extensions Contains additional parameters requesting additional processing by the client and authenticator. Registered extensions are listed in the [IANA-WebAuthn-Registries].

```
authenticator_selection_criteria = {
    ? 1 => uint .size 1,            ; authenticator_attachment,
                        mapped to enum of AuthenticatorAttachment
    ? 2 => uint .size 1 .default 1, ; resident_key,
                        mapped to enum of ResidentKeyRequirement
    ; requireResidentKey from Webauthn L2 is missing,
    ; because this extension does not require
    ; compatibility with WebAuthn L1
    ? 3 => uint .size 1 .default 2  ; user_verification,
                        mapped to enum of UserVerificationRequirement
]
```

authenticator_attachment: Relying Parties use this value to express a preferred authenticator attachment modality for the creation of the new credential.

resident_key: Describes the Relying Parties requirement for client-side discoverable credentials.

user_verification: Describes the Relying Parties requirement regarding user verification for the creation of the new credential.

```
enum {
    platform(1),
    cross-platform(2)
} AuthenticatorAttachment
```

```
enum {
    discouraged(1),
    preferred(2),
    required(3)
} ResidentKeyRequirement
```

```
enum {
    required(1),
    preferred(2),
    discouraged(3)
} UserVerificationRequirement
```

```
enum {
    usb(1),
    nfc(2),
    ble(3),
    internal(4)
} AuthenticatorTransport
```

```
enum {
    none(1),
    indirect(2),
    direct(3),
    enterprise(4)
} AttestationConveyancePreference
```

```
extensions = {
    extension_id => extension_data,
]
extension_id = tstr .size (1..256)
extension_data = bstr .size (1..2^12)
```

12.5. Registration Response Message

```
RegistrationResponse = [
    message_type: 5,
    attestation_object: bstr .size (1..2^13),
    clientdata_json: tstr .size (1..2^11)
]
```

message type: An identifier used for easier message processing and debugging.

attestation_object: The attestation object is created by the Authenticator. It contains authenticator data and an attestation statement, as well as additional information so that the Relying Party can validate the attestation statement, authenticator data and clientdata_json. That way the Relying Party can verify the integrity of the newly created public key credential. It is opaque to the client. Defined in [WebAuthn], Section 6.5.

clientdata_json: JSON-serialized string containing the collected client data. The hash of the clientdata_json is used by the authenticator. Defined in [WebAuthn], Section 5.2.

12.6. Authentication Indication Message (discoverable credential)

```
AuthenticationIndicationDiscoverable = [
    message_type: 6;
]
```

message type: An identifier used for easier message processing and debugging. In the ClientHello message it also serves the RP to identify the client's desired action.

12.7. Authentication Indication Message (server-side credential)

```
AuthenticationIndicationServerSide = [
    message_type: 7,
    ephemeral_user_id: bstr .size 256,
    encrypted_user_name: bstr .size 256
]
```

message type: An identifier used for easier message processing and debugging. In the ClientHello message it also serves the RP to identify the client's desired action.

ephemeral_user_id: An ephemeral reference that is used to link the PreHandshake and the second handshake. Received by the client from the RP in the PreHandshake.

encrypted_user_name: Padded, and encrypted user name as described in Section 8. The user name is a human-palatable identifier for the user account. The user name is defined in [WebAuthn], Section 5.4.1.

12.8. Authentication Request Message

```
AuthenticationRequest = [
    message_type: 8,
    challenge: bstr .size (16..64),
    ? optionals
]
```

message type: An identifier used for easier message processing, and debugging. In the ClientHello message it also serves the RP to identify the client's desired action.

challenge: Challenge that MUST be randomly generated by the Relying Party in an environment they trust. Signed by the authenticator when producing the authentication assertion. Defined in [WebAuthn], Section 13.4.3.

optionals: Optional parameters. Defined in [WebAuthn], Section 5.5.

```
optionals = {
    ? 1 => int .size 4                          ; timeout
    ? 2 => tstr .size (1..256)                  ; rp_id
    ? 3 => [* public_key_credential_descriptor] ; allow_credentials
    ? 4 => int .size 1 .default 2               ; user_verification
    ? 5 => extensions                           ; extensions
}
```

timeout: Time in Milliseconds that the caller is willing to wait for the call to complete. MAY be overridden by the client.

rp_id: Relying Party Identifier claimed by the caller. If omitted, set to the server's origin's effective domain.

allow_credentials: List of credentials acceptable to the caller. Ordered from most preferred to least preferred. The list is encrypted as described in Section 8. **MUST** be present with at least one element if server-side credentials are used. **MUST NOT** be present if discoverable credentials are used.

user_verification Describes the RPs requirement regarding user verification.

extensions Contains additional parameters requesting additional processing by the client, and authenticator. Registered extensions are listed in the [IANA-WebAuthn-Registries].

```
public_key_credential_descriptor = [
type: uint .size 1,                ; mapped to enum of
                                     PublicKeyCredentialType
    id: bstr .size (16..256),
    ? transports: [* uint .size 1] ; mapped to enum of
                                     AuthenticatorTransport
]
```

type: Describes valid credential types.

id: Contains the credential ID of the public key credential.

transports: Optional parameter. List of hints as to how clients might communicate with a particular authenticator. Represents the RPs best belief as to how an authenticator might be reached.

```
enum {
    public-key(1)
} PublicKeyCredentialType
```

```
enum {
    usb(1),
    nfc(2),
    ble(3),
    internal(4)
} AuthenticatorTransport
```

```
enum {
    required(1),
    preferred(2),
    discouraged(3)
} UserVerificationRequirement
```

```
extensions = {
    * extension_id => extension_data,
]
extension_id = tstr .size (1..256)
extension_data = bstr .size (1..2^12)
```

12.9. Authentication Response Message

```
AuthenticationResponse = [
    message_type: 9,
    clientdata_json: bstr .size (1..2^11),
    authenticator_data: bstr .size (37..512),
    signature: bstr .size (1..32),
    ? user_handle: bstr .size 64
]
```

message type: An identifier used for easier message processing, and debugging.

clientdata_json: JSON-serialized string containing the collected client data. The hash of the clientdata_json is used by the authenticator. Defined in [WebAuthn], Section 5.2.

authenticator_data: Byte array that encodes contextual bindings made by the authenticator. Includes a hash of the RP ID, flags, and a signature counter. Optionally, includes attested credential data defined in [WebAuthn], Section 6.5.1, and extension-defined authenticator data. The authenticator data is defined in [WebAuthn], Section 6.1.

signature: Raw signature returned from the authenticator. The procedure how this signature is generated is defined in [WebAuthn], Section 6.3.3.

user_handle: Optional parameter. Contains the user ID returned from the authenticator. MUST be set when discoverable credentials are used.

13. Implementation Status

Note to RFC editor: Remove this section, as well as the reference to [RFC7942] before publication

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

There are two prototype proof-of-concept implementations available. The implementations were done before the draft was finished and is therefore not compatible with any draft version (structure definitions can differ, the PreHandshake was changed), but the overall concept is proven, as well as the viability of the double handshake approach. The source code of the implementation in C can be found under https://github.com/wede-kind/fidoSSL. The source code of the implementation in Rust can be found under https://github.com/7ritn/rustls-fido.

14. IANA Considerations

This document has IANA actions:[RFC8446] [RFC8447]:

- Entry of the "fido2" extension in the TLS ExtensionType Values Registry
- Entry of the "fido2_error" alert in the TLS Alert Registry.

The "Value", and "Registry" values are not final.

Value	Extension Name	TLS 1.3	DTLS-Only	Recommended	Reference
63	fido2	CH, CR, CT	N	Y	I-D FIDO2 TLS 1.3 Extension

Table 1: Addition to the TLS ExtensionType Values Registry

Value	Description	DTLS-OK	Reference
122	fido2_error	Y	I-D FIDO2 TLS 1.3 Extension

Table 2: Addition to the TLS Alert Registry

15. Security Considerations

This extension introduces new risks for security and especially forward security if the server does mistakes. The client has no direct way through the extension to know if the server does everything correctly. This includes no reuse of ephemeral user IDs and 256-bit keys by the server, as well as secure generation of ephemeral user IDs and 256-bit keys. For best possible forward security, the server has to delete the 256-bit keys, or change the master secret after a reasonable time span

Servers that hold information between the two handshakes for too long could leave themselves open for DoS attacks.

The security properties of FIDO are described in [FIDO-SecRef].

16. References

16.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8447] Salowey, J. and S. Turner, "IANA Registry Updates for TLS and DTLS", RFC 8447, DOI 10.17487/RFC8447, August 2018, <https://www.rfc-editor.org/info/rfc8447>.

[WebAuthn] World Wide Web Consortium, "Web Authentication: An API for accessing Public Key Credentials Level 2", 8 April 2021, <https://www.w3.org/TR/2021/REC-webauthn-2-20210408/>.

[FIDO-CTAP2] FIDO Alliance, "Client to Authenticator Protocol (CTAP)", 21 June 2022, <https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-errata-20220621.html>.

[FIDO-SecRef] FIDO Alliance, "FIDO Security Reference", 23 May 2022, <https://fidoalliance.org/specs/common-specs/fido-security-ref-v2.1-ps-20220523.html>.

[IANA-WebAuthn-Registries] IANA, "Web Authentication (WebAuthn) registries", <https://www.iana.org/assignments/webauthn/>.

[HTML-Living-Standard] Web Hypertext Application Technology Working Group, "HTML Living Standard", 5 February 2026, <https://html.spec.whatwg.org/multipage/browsers.html>.

16.2. Informative References

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

[FIDO-Glossary] FIDO Alliance, "FIDO Technical Glossary", 23 May 2022, <https://fidoalliance.org/specs/common-specs/fido-glossary-v2.1-ps-20220523.html>.

[I-D.ietf-emu-eap-fido] Rieckers, J. and S. Winter, "EAP-FIDO", Work in Progress, Internet-Draft, draft-ietf-emu-eap-fido-01, 3 March 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-emu-eap-fido-01>.

[HTML-living-standard-origin] Web Hypertext Application Technology Working Group, "HTML Living Standard", 23 January 2026, <https://html.spec.whatwg.org/multipage/browsers.html#origin>.

Appendix A. Open Questions

Note to RFC Editor: Remove this section and all references from this section before publication.

A.1. How to determine and validate the scheme of the origin?

Determining the scheme of the origin provides more difficult than the host, as outside the web context, the concept of a scheme can not be easily applied. TLS is not bound to any specific application protocols, and has no knowledge of what specific application is transporting data over it. Possible ideas are:

- The scheme could be a configurable item on the side of the client, and server. How the scheme is provided by the server, and validated by the client, is still uncertain.
- As the scheme "https" is mandatory in WebAuthn, it could be adopted for every origin, independent of the actual application. This could have unpredictable implications.
- The scheme could possibly also be dropped completely. The client would then have to rely solely on the host and RP ID to determine the server identity. This could deteriorate security.

Although not concerning the scheme itself, the problem of determining the RP ID is also discussed in the Internet-Draft "EAP-FIDO" [I-D.ietf-emu-eap-fido]. The context is a bit different, but maybe some of its ideas could be useful in finding a proper solution.

Author's Address

David Wedekind (EDITOR)
Humboldt-Universität zu Berlin
Email: wedekind@hu-berlin.de

B. Wireshark Dissector Guide

This section provides instructions on how to set up and test the Wireshark dissector presented in 6.2.

B.1. Prerequisites

- A Wireshark installation supporting any LUA version from 5.1 to 5.4.
- A working installation of the fidoSSL version created for this thesis. Instructions for that can be found here:`https://github.com/wede-kind/fidoSSL`. For the purpose of this guide, the `client` and `server` test applications are assumed to be located in the *fidoSSL/build* directory.

B.2. Installation of the Wireshark plugin

- Place the files `FIDO2_TLS_dissector.lua` from `https://github.com/wede-kind/fido-tls-dissector` and the file `cbor.lua` from `https://github.com/Kristopher38/lua-cbor`[36] into the Wireshark plugin directory. On Unix-like systems the standard location is */.local/lib/wireshark/plugins*.

B.3. Testing the Wireshark plugin

- Open two terminals, one for the test server and one for the test client with the *fidoSSL* directory being the working directory.
- In the terminal for the client, set the `SSLKEYLOGFILE` environment variable to a valid location, where the file containing the SSL/TLS session keys should be stored. For example:

  ```
  export SSLKEYLOGFILE=~/temp/sessionkeys.log
  ```

- In Wireshark, start a capture for localhost, called `Loopback: lo` on some Unix-like systems.
- Start the test server by executing `./build/server` in its terminal.
- Start the test client by executing `./build/client` in its terminal.
- After the FIDO registration or authentication is finished, stop the Wireshark capture.
- The key-log file can now be integrated into Wireshark, by following in the Wireshark application: `Edit->Preferences`. In the dialog box that opens, navigate to `Protocols->TLS`. The path to the key-log file can then be inserted into the `(Pre)-Master-Secret log filename`.

- In the capture, the packets can now be filtered for packets containing the FIDO2 Extension by inputting `tls.handshake.extension.type == 4660` into the filter field.
- The dissected content of the FIDO2 Extension can now be found in the entry `FIDO Data` in the inspection tab of the shown packets. Examples are shown on the following screenshots.

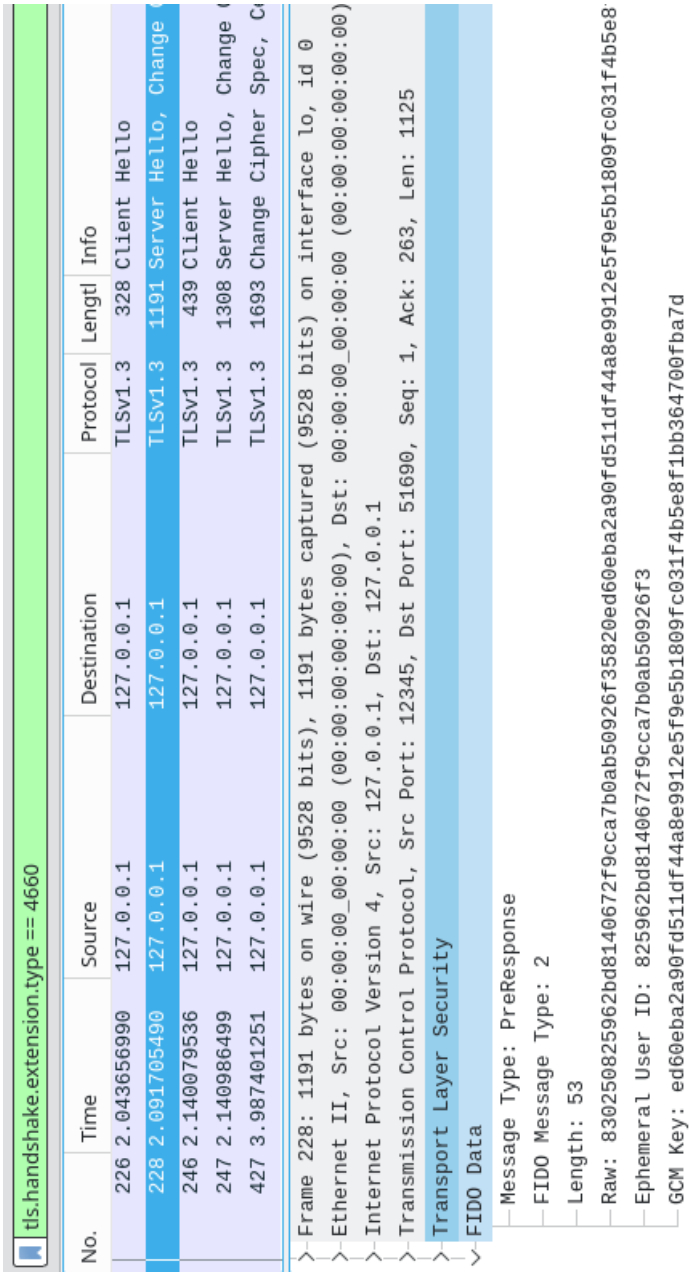

Figure 8: Screenshot of a Wireshark Packet Dissection showing a FIDO2 Extension PreRequest. Created by author

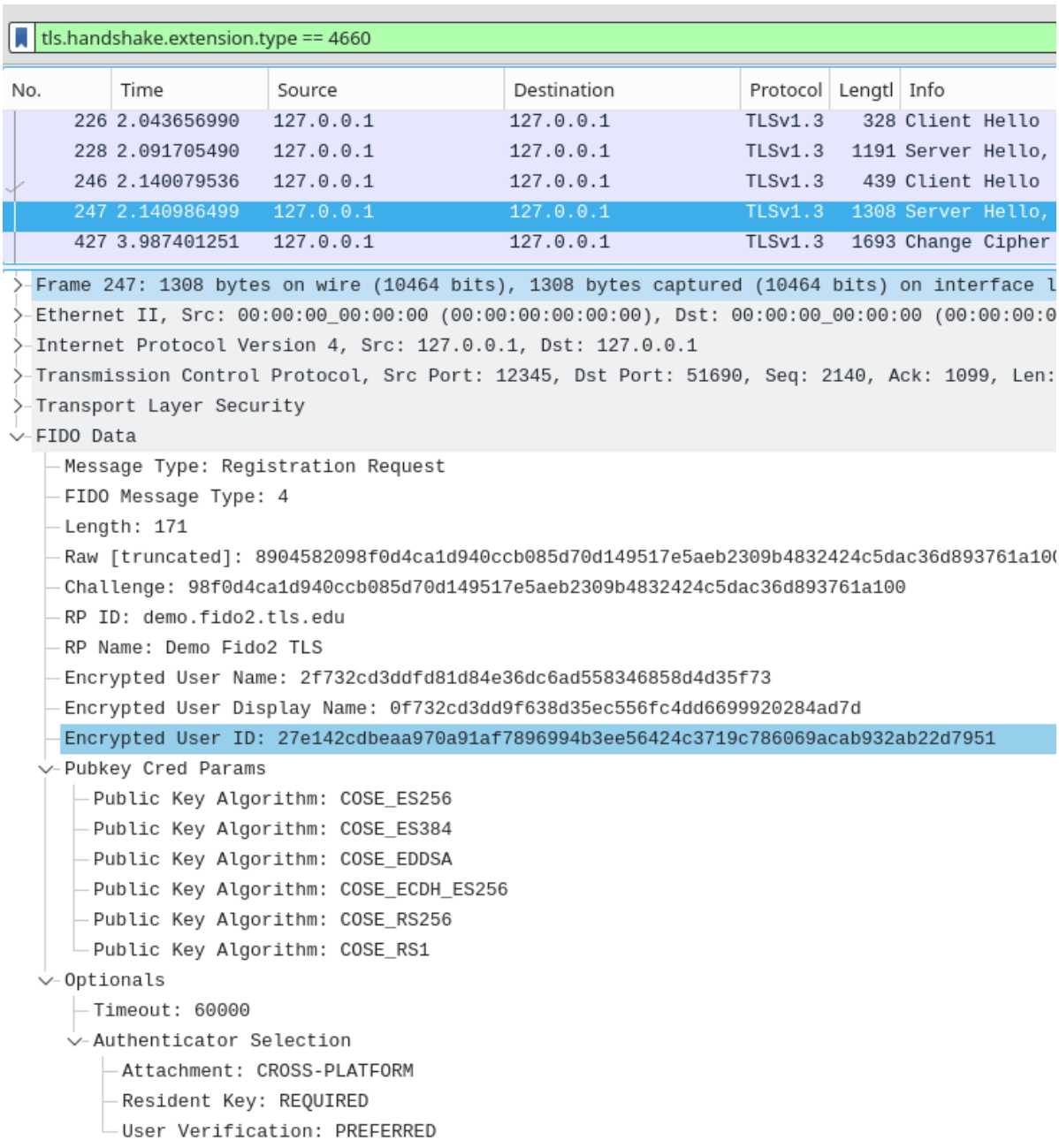

Figure 9: Screenshot of a Wireshark Packet Dissection showing a FIDO2 Extension Registration Request. Created by author

Figure 10: Screenshot of a Wireshark Packet Dissection showing a FIDO2 Extension Authentication Response. Created by author

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 23. Februar 2026 [redacted]