FIDO2 – What can a man in the browser do?

Analysis of the strength of a man in the browser against the fido2 authentication mechanism.

Course
Project 2

Author
Cyrill Bolliger

Supervisor
Prof. Gerhard Hassenstein

Date
June 1, 2020

Bern University of Applied Sciences
Department of Engineering and Information Technology

Abstract

This paper examines what harm a malicious browser extension can do to a FIDO2 authentication. Many investigations on the security of FIDO2 have been published but none of them scrutinizes browser extensions explicitly. However, browser extensions are fairly common and extremely powerful. Therefore, this study analyzes the threat of a malicious browser extension in theory and fortifies the findings with a practical proof of concept implementation. It identifies a rogue browser extension as a serious threat to the level of assurance of FIDO2 as the man-in-the-browser is able to register a forged public key and subsequently authenticate without user interaction at any time. Hence, the suitability of FIDO2, without an additional out of band verification, as authentication method in a high security context is questioned.

Contents

1. Introduction

Strong user authentication is the foundation of numerous applications. Best known from some online services like webmail, user authentication gets even more important with the introduction of legally binding electronic signatures, digital health records etc. The classic authentication method with username and password suffers many vulnerabilities and human weaknesses. It is therefore not suited as sole authentication factor in a context with elevated security needs (Grassi et al. 2017).

This is where the FIDO2 standard comes into play. It is suited for authenticators of all National Institute of Standards and Technology (NIST) assurance levels (Grant et al. 2017). Yet, it is not only considered as standard for strong authentication but also aims to provide an excellent user experience (FIDO Alliance 2020).

FIDO2 has been scrutinized many times out of different perspectives. The closest to this paper is probably the excellent examination of Stötzner (2018), who identified FIDO2 to be susceptible to a man-in-the-middle attack even if an authenticator attestation is required. However, no studies dedicated to a man-in-the-browser attack were found. Therefore this paper seeks to analyze the strength of a man-in-the-browser in the form of a browser extension. After a theoretical analysis of possible attack vectors, a proof of concept implementation verifies it's feasibility.

A malicious browser extension is a very powerful adversary, yet such extensions happened to appear in the official browser extension stores multiple times in the past (Brinkmann 2018; Goodin 2018). It is therefore considered a real threat.

This work is structured as follows: In chapter 2 the FIDO2 standard is introduced and the registration and authentication flows are explained. chapter 3 then identifies and describes possible attack vectors of a browser extension. To demonstrate their practicability, a proof of concept browser extension was implemented and is described in chapter 4. chapter 5 finally discusses the findings, tries to put them into relation with other threats and suggests further research.

FIDO2 consists out the Client to Authenticator Protocol (CAPT) and Web Authentication Protocol (WebAuthN). This study solely focuses on the WebAuthN. Attacks to the CAPT as well as threats different to man-in-the-browser attacks, are not subject to this study.

As WebAuthN is a subset of FIDO2, concerns that apply to WebAuthN do also apply to FIDO2. The two terms were used accordingly: FIDO2 sometimes refers to WebAuthN, but WebAuthN never refers to something else than itself.

2. FIDO2

FIDO2 is a standard for simple yet strong authentication based on public key cryptography. Developed by the FIDO Alliance in cooperation with the World Wide Web Consortium (W3C), it is backed by many big players like Amazon, Google, Facebook etc. It consists out of the WebAuthN and the CAPT protocols (FIDO Alliance 2020). With the first public working drafts of both specifications released in 2016, the W3C recommendation for WebAuthN released in 2019 and the CAPT specification still being in the state of a *proposed* standard, FIDO2 is bleeding edge technology (Balfanz et al. 2019; Brand et al. 2019).

To *authenticate* using FIDO2, one must first *register* his credential public key. As it is crucial to understand those two processes in order to discuss possible attacks, they will be briefly explained in a simplified way in the rest of this chapter. Readers interested in more detail are invited to study the specifications.

2.1 Terminology

FIDO2 uses specific terminology, which is well defined in the specification. The most important among them will, however, be briefly introduced here.

Assertion An object containing some parameters and a cryptographic signature that confirms the possession of a specific private key. The assertion actually proofs the authentication claim.

Attestation An object containing some parameters and optionally a cryptographic signature that can proof the genuineness of the authenticator.

Authenticator The cryptographic device, that generates the credential's key pair and performs all actions that require the private key. The authenticator can be an external device or use a built in Trusted Platform Module (TPM).

Client The endpoint for the WebAuthN and the CAPT protocol. Usually, this is the browser.

Relying Party The actor, that requests the user to authenticate. This is typically a server of a web-service.

2.2 Registration

As shown in figure 2.1, the client initiates the registration process (1). The relying party responds with a challenge and some parameters for the credential creation (2). The client then passes those parameters on to the authenticator (3). After the user proofed it's presence by interacting with

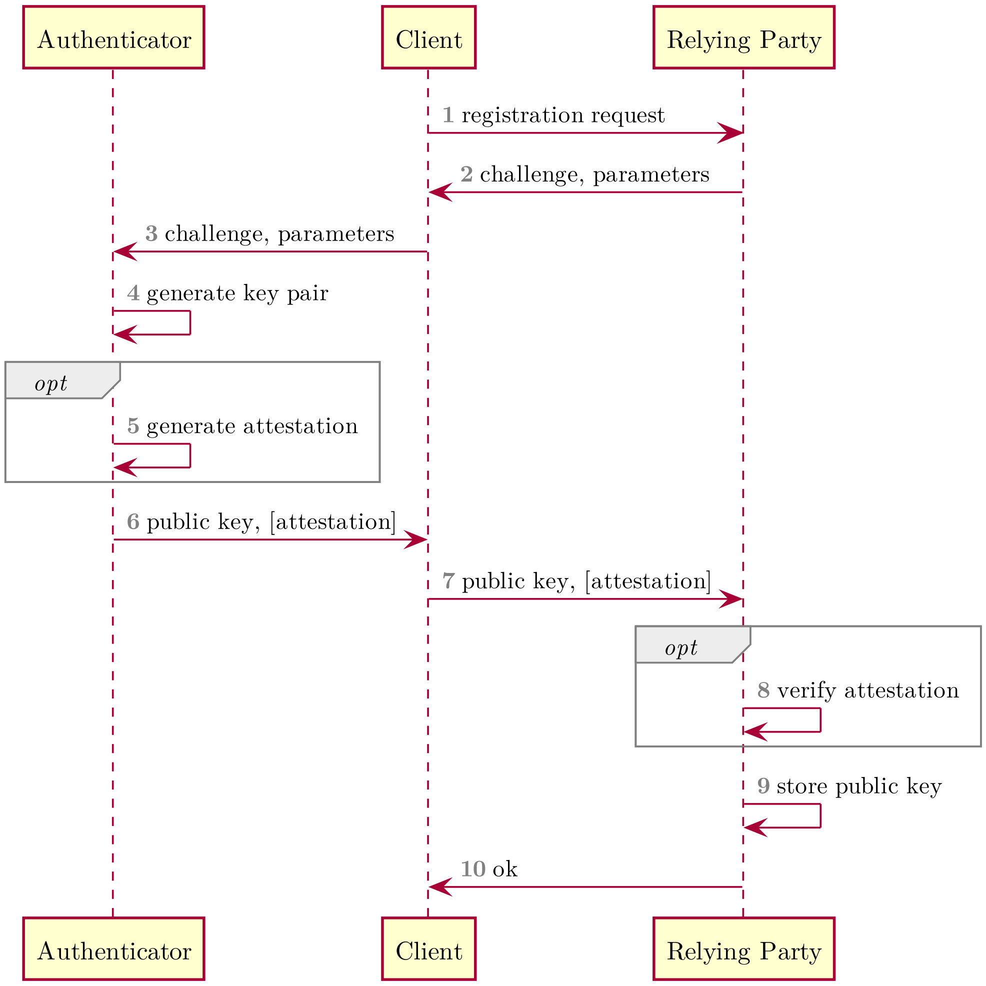

Figure 2.1: FIDO2 registration flow

the authenticator, the authenticator generates a new key pair (4). If specified in the credential creation parameters, the authenticator signs some data about the authenticator, the public key and the data received from the relying party as attestation (5). It then sends the attestation, together with the public key, back to the relying party via the client (6, 7). The relying party optionally checks the signature of the attestation (8) before storing the public key as credential.

2.3 Authentication

The authentication process, depicted in figure 2.2, is closely related to the registration process explained in section 2.2. Instead of generating a key pair, the private key corresponding to the previously registered public key is used to sign the challenge (together with some other parameters) in order to generate the assertion (4). The relying party then uses the public key stored at registration time, to verify the assertion (7).

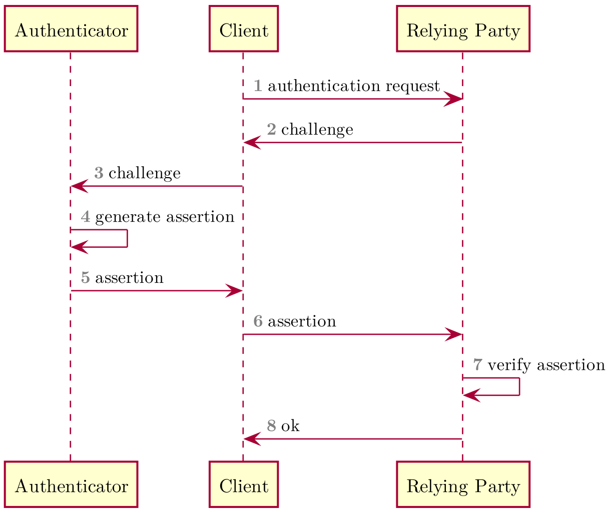

Figure 2.2: FIDO2 authentication flow

2.4 Attestation

As the attestation will be important in the next chapter, we'll have a closer look at it. Prior to going in to more detail, it is important to distinguish between the attestation statement and the attestation object. The attestation object contains the attestation statement alongside the credentials public key and some other data. In this paper, the colloquial term attestation is used to refer to the attestation object, however sometimes with a strong focus on the attestation statement.

Several attestation types exist. They can be partitioned in *none*, *self* and *vendor*, while none contains the attestation type *none*, self the attestation type *self* and vendor contains the remaining attestation types (*basic*, *CA*, *ECDSA*).

The attestation statement contains a cryptographic signature of the credentials public key and the authenticator itself. The signature is created with "the key of the attesting authority (except for the case of self attestation, when it is created using the credential private key)" (Balfanz et al. 2019, sec. 6.4). If attestation type *none* is used, the attestation statement is omitted.

3. Attack Vectors

The current WebAuthN recommendation states that the registration process is one of the weak spots in FIDO2, if attestation type *none* or *self* is used.

> [. . .] it is possible for a man-in-the-middle attacker – for example, a malicious client or script – to replace the credential public key to be registered, and subsequently tamper with future authentication assertions scoped for the same Relying Party and passing through the same attacker [. . .] Note, however, that such an attack would be easy to detect and very difficult to maintain, since any authentication ceremony that the same attacker does not or cannot tamper with would always fail. — Balfanz et al. 2019, sec. 13.3.1.

The following sections, of this paper will first proof the quoted assumption to be extremely optimistic and secondly outline a man-in-the-middle attack for the other attestation types.

This paper will extensively refer to the security assumptions (SA) and security goals (SG) of FIDO2, as they can be found in Lindemann et al. (2017). For the readers convenience, they are provided in section A.1 and section A.2 respectively.

3.1 No / Self Attestation

If we assume, the man-in-the-middle would be a malicious browser extension, called man-in-the-browser, the attack works as depicted by figure 3.1 and figure 3.2, detailed below.

3.1.1 How the attack works

Following the sequence diagram in figure 3.1 we can not locate any anomalies in the registration process until step 7. In step 7 however, the man-in-the-browser comes into play. He intercepts the public key, that the client would normally send to the relying party and drops it (7). The man-in-the-browser then generates a rogue key pair *key pair*$'$ (8) and optionally an attestation *attestation*$'$, which he signs with the rogue private key (9). He then sends the rouge *public key*$'$ together with the *attestation*$'$ to the relying party (10). As the man-in-the-browser uses the same connection as the client would use, the relying party is unable to detect the attack. The relying party just verifies the *attestation*$'$ (11) and stores the *public key*$'$ as if everything would be normal.

The attack in the authentication flow, shown in figure 3.2 does not differ much from the registration: The man-in-the-browser intercepts the assertion (6) and replaces it with the new assertion *assertion*$'$ (7), signed with the rouge private key generated in the registration flow. When the relying party receives the *assertion*$'$ (8) it verifies it against the rouge *public key*$'$ (9) stored in the registration phase, which will, unsurprisingly, succeed.

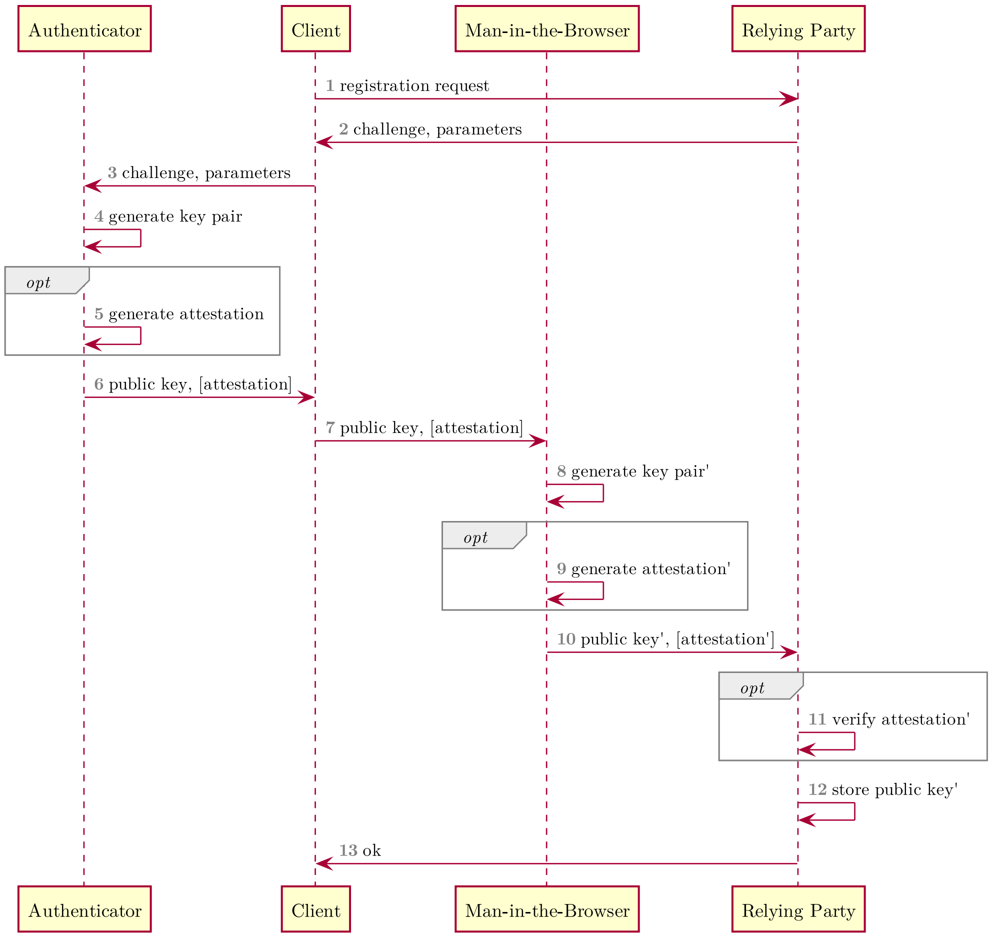

Figure 3.1: FIDO2 registration flow with man-in-the-browser

3.1.2 Security Considerations

With the attack outlined above, the man-in-the-browser is capable to authenticate at any time, as long the the browser is running, without the user noticing anything. It violates the security assumption SA-4 and is capable to break the security goals SG-7, SG-11, and SG-13 (cf. section A.1 and section A.2).

In the current WebAuthN recommendation Balfanz et al. state, that "such an attack would be easy to detect and very difficult to maintain" (2019). I do contradict this claim for the following reasons:

1. Browser extensions are typically installed for an extended time period. It is therefore not unlikely, that a user registers a new FIDO2 token during this time frame. As long as the extension stays installed, the attack can easily be maintained.

2. Google Chrome, which is by far the most popular browser in 2020 for desktop (statscounter 2020a) and mobile (statscounter 2020b) offers users to synchronize the installed browser extensions and their data across devices. Using `storage.sync` a browser extension can easily discover, if a user has enabled this feature and only then mount the attack. As long as the

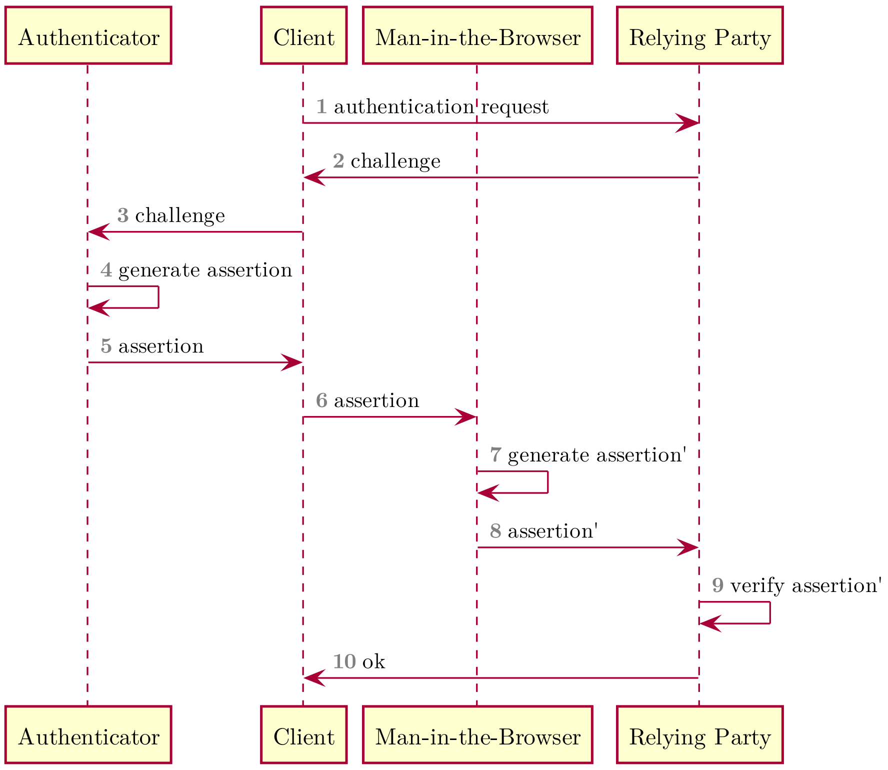

Figure 3.2: FIDO2 authentication flow with man-in-the-browser

user uses any of his synchronized devices and his regular browser – which he will likely do for sensitive actions – he won't notice anything.

3. Publishing malicious browser extensions in the official webstore is not impossible. We have seen it several times in the past already (Brinkmann 2018; Goodin 2018).

3.2 With Attestation

If the relying party requires us to provide a vendor attestation (cf. section 2.4), the attack gets slightly more complicated. We have to introduce an additional entity, that will subsequently be called *accomplice*. The accomplice is a remote server in the control of the attacker. It has genuine authenticators of multiple vendors plugged in, each of them equipped with a robot that proofs user presence and user verification, when needed.

Please note, that the figure 3.3 and figure 3.4 are slightly simplified, as the authenticators of the accomplice are not drawn as separate entities.

3.2.1 How the attack works

As shown in figure 3.3 the man-in-the-browser eavesdrops the challenge received from the relying party (2). It then sends it to the accomplice (3). The accomplice doesn't check the origin and simply performs the regular registration process with it's plugged in authenticator (4, 5). It then returns the *public key′* and the *attestation′* to the man in the browser (6).

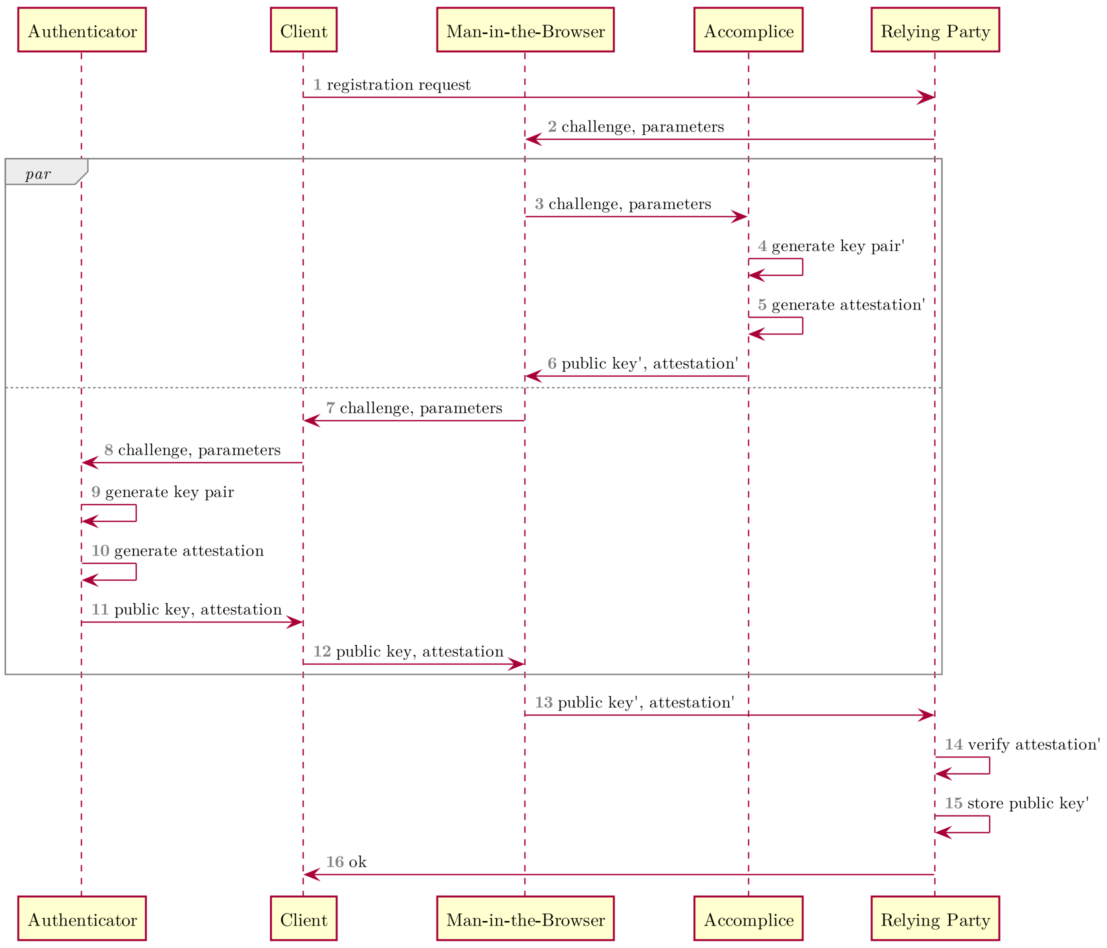

Figure 3.3: FIDO2 Attested registration flow with man-in-the-browser

Meanwhile the legitimate registration process of the victim continues as normal (8, 9, 10, 11). Afterwards, the man-in-the-browser drops the legit *public key* and *attestation* (12) and instead sends the *public key′* and *attestation′* generated by the accomplice to the relying party (13). The relying party checks the accomplice's *attestation′*, which comes from a genuine device and is therefore valid (14). It then stores the accomplice's *public key′* (15) and confirms the registration (16).

The authentication (cf. figure 3.4) is not much different: The man-in-the-browser again eavesdrops the challenge (2) and lets the accomplice generate the *assertion′* (3, 4, 5). Meanwhile the victim also conducts the necessary steps to generate an *assertion* (7, 8, 9). The man-in-the-browser then drops the legit *assertion* and instead sends the *assertion′* of it's accomplice to the relying party. The *assertion′* verifies, as it was generated with the corresponding authenticator (12). The relying party finally confirms the successful authentication (13).

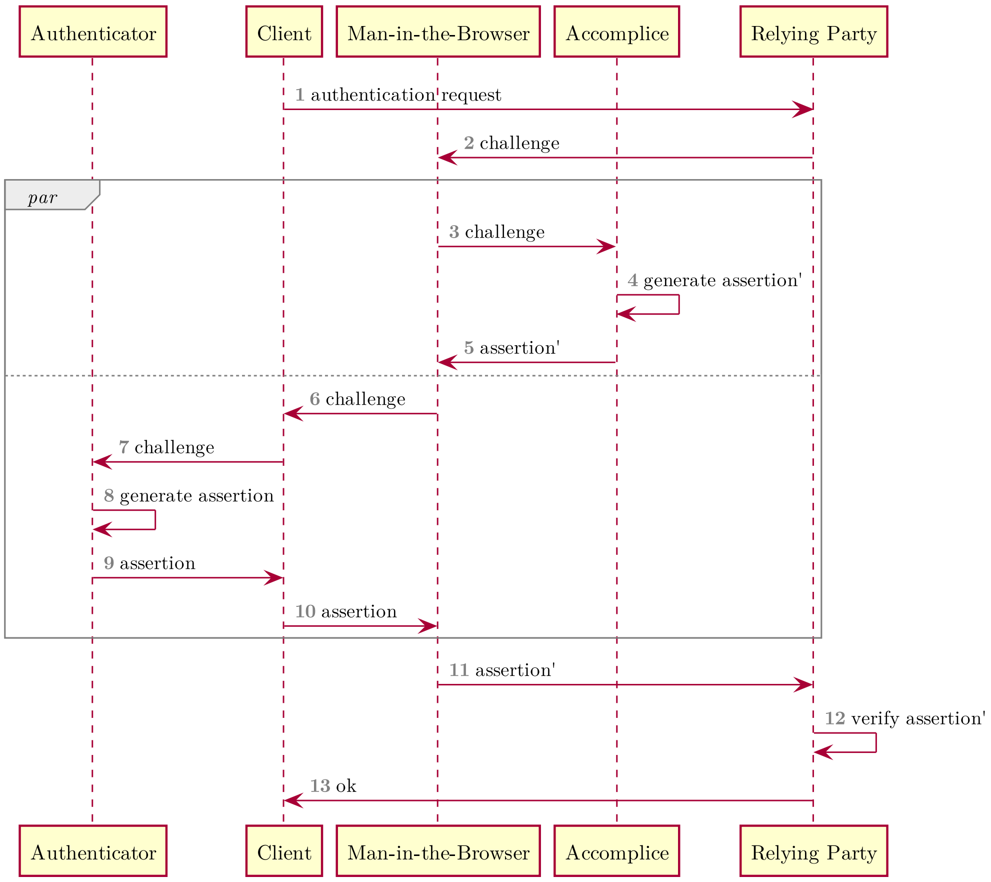

Figure 3.4: FIDO2 Attested authentication flow with man-in-the-browser

3.2.2 Security Considerations

Even if the relying party limits its use to authenticators that are able to provide a particular attestation, FIDO2 is susceptible to man-in-the-middle attacks. Unfortunately the current WebAuthN recommendation doesn't mention this vulnerability, but rather implies, that only the attestation types *none* and *self* suffer this weakness (cf. Lindemann et al. 2017, sec. 13.3.1). Thanks to the findings of Stötzner (2018) the latest editors draft of WebAuthN fixes this issue.

With the attack described, the security goals SG-7, SG-11, SG-13 and SG-14 are suspended, due to the violation of the security assumption SA-4 (cf. section A.1 and section A.2).

Mounting such an attack is more complex than the attack outlined in section 3.1, but with the necessary engineering skills to build the robots, that proof user presence and verification, it is still considered feasible.

4. Proof of concept

As shown in chapter 3, in theory a man in the middle attack on FIDO2 is possible. To proof this theory, a minimal working example for an attack, as it was described in section 3.1, was implemented. It was designed as browser extension for Firefox, that intercepts registration and authentication messages against the FIDO2 demo implementation published on webauthn.io.

A general overview, on how the extension works, can be found in section 3.1. The following section provides some insights into selected parts of the code. The full source code is published on github.com/cyrillbolliger/fido2.

4.1 The Browser Extension

The browser extension is written in JavaScript and bundled with webpack. It evedrops the ajax communication between the browser and webauthn.io using the WebExtension API:

```
/**
 * This is the entry point to intercept responses to the RP.
 */
browser.webRequest.onBeforeRequest.addListener(
    listener,
    {urls: ['https://webauthn.io/*'], types: ['xmlhttprequest']},
    ['requestBody', 'blocking']
);
```

Listing 4.1: request-handler.js

The listener then checks, if the message matches the format of a WebAuthN response, decodes the credential data and modifies the credential subsequently:

```
modifyCredential(credential, url) {
    const originalCredential = cloneDeep(credential);
    const evilCredential = credential;
    let promise;

    if (originalCredential.response instanceof
       MyAuthenticatorAttestationResponse) {
        console.log("It's an AuthenticatorAttestationResponse.
           Generating evil keys.");
        promise = credential.generateEvilKeys()
            .then(() => evilCredential.saveEvilKeys())
            .then(() => evilCredential.replaceKeys())

            // output evil PK in PEM format for easy validation later on
            .then(() => evilCredential.getEvilPubKeyPem())
            .then(pk => console.log(`The evil public key is:\n${pk}`));

    } else if (originalCredential.response instanceof
        MyAuthenticatorAssertionResponse) {
        console.log("It's an AuthenticatorAssertionResponse. Signing
            with evil key.");
        // todo: handle reject case (the key was not found)
        promise = evilCredential.loadEvilKeys()
            .then(() => evilCredential.signWithEvilKeys());

    } else {
        // do nothing and the request will pass as it is
        return;
    }

    promise.then(() => {
        const evilBody = evilCredential.encode();
        const originalBody = originalCredential.encode();

        console.log('PublicKeyCredential successfully modified.');
        console.log('original PublicKeyCredential:\n', originalBody);
        console.log('evil PublicKeyCredential:\n', evilBody);
        console.log('Sending evil PublicKeyCredential to relying party.'
            )

        fetch(url, {
            method: 'POST',
            body: stringToArrayBuffer(evilBody),
        }).then(resp => {
            console.log('The relying party responded:\n', resp);

            if (200 === resp.status) {
                browser.tabs.update({url: redirectUrl});
                console.log('Login with evil key successful.');
            } else if (201 === resp.status) {
                console.log('Evil key successfully registered.');
                // todo: use the response to fake the changes in the dom
                // that a non-intercepted response would trigger
            } else {
                console.log('Attack failed.')
            }
        });
    });

    // todo: store original request but cancel it and if promise
    // resolves, dump original request. else resend it

    // drop request
    return {cancel: true};
}
```

Listing 4.2: request-handler.js

If the intercepted response is a registration request, a new key pair is generated, stored in browser storage and finally injected into the credential (line 62–71). Else, if the response is an authentication, the browser extension loads the evil key pair generated on registration and uses it to replace the assertion's signature (line 72–77).

Since the browser doesn't allow to modify the original request's body, a new request with the rouge credential is launched (line 92–95). As the request now originates from the extension instead of the page in the browser, we have to handle the response manually (line 95–109). Finally, the original request is dropped, so it will never be seen by the relying party (line 115).

To inject the evil public key, the registration message was decoded, the key replaced, and the whole object re-encoded. All other properties are left as they are.

This whole process of encoding and decoding is fairly complex and it would exceed the scope of this document, to go into the details. Interested readers are advised to study the `encode()` and `decode()` methods of the classes

- `credentials/MyAuthenticatorAttestationResponse.js`
- `credentials/MyAttestationObject.js`
- `credentials/MyAuthenticatorData.js`
- `credentials/MyAttestedCredentialData.js`

as well as the `jwkToCose` module in `util/jwk-to-cose.js`, which had to be crafted explicitly for this browser extension, as no library containing this functionality could be found.

Once the registration has completed, the browser extension may listen for authentication messages. If a message between the browser and the relying party matches the format of an assertion and an evil key for the corresponding relying party was registered, the assertion is replaced.

Apart from some de- and encoding, replacing the signature is trivial:

```
async sign(evilKeys) {
    const data = await this._getSignatureInput();
    const signature = await crypto.subtle.sign(evilKeys.getAlgos(),
        evilKeys.privateKey, data);
    this.signature = this.encodeSignature(signature);
}
```

Listing 4.3: credential/MyAuthenticatorAssertionResponse.js

As on registration, we then drop the original request and send the rogue assertion. Again, the response isn't automatically handled by the browser, but the extension jumps in (listing 4.2, line 99), so the user won't notice anything.

4.2 Challenges

Due to the fact, that the browser extension has to fake the Document Object Model (DOM) modification initiated by a FIDO2 response (or redirect the client), a dedicated attack has to be crafted for any targeted service. If multiple services should be attacked, the extension must thus have an implementation for every single one.

While testing the browser extension against different FIDO2 servers, it was also discovered, that the message formats differ, depending on the server. The non-uniform message format further complicates a general attack against every server but doesn't render a targeted attack more difficult.

4.3 Limitations

FIDO2 supports a multitude of algorithms, modes and extensions. As the implemented browser extension is only a proof of concept, no FIDO2 extensions are supported. Also in terms of algorithms, the browser extension is limited to Elliptic Curve Digital Signing Algorithm (ECDSA) signatures on the P-256 curve, based on SHA-256 hashes.

The attestation mode is limited to `none`, even thou support for the mode `self` could be implemented easily. For other attestation modes, an accomplice server with genuine authenticator devices is required, as outlined in section 3.2.

An other limitation applies to the browser. The developed browser extension was crafted to work with Firefox. Thanks to the mostly compatible WebExtension API, adapting it to Chromium based browsers would, however, not require major changes (MDN contributors 2020).

4.4 Variations

Looking at listing 4.2, it is clearly visible, that an attack with attestation as outlined in section 3.2, would also be implementable. Instead of generating the public key or signature itself, the browser extension would let the accomplice perform this tasks and replace the genuine FIDO2 response of the local token with the rogue response from the accomplice. So the relying party would finally not receive the FIDO2 message of the user's FIDO2 device, but of the device from the accomplice instead.

5. Discussion

It's not new, that FIDO2 is susceptible to man-in-the-middle attacks, as it was already mentioned in the WebAuthN recommendation (Balfanz et al. 2019). As Stötzner pointed out, an attack is even possible if an attestation type other than *none* or *self* is requested (2018). The discussion of this issue withal focuses on broken Transport Layer Security (TLS) sessions and malicious scripts, especially in the context of Cross Site Scripting (XSS) (Stötzner et al. 2018). This implies, that maintaining the attack for a long time, in order to not be detected, is hard. However, this doesn't apply for a browser extension. A rouge browser extension doesn't depend on any weaknesses in TLS nor any XSS vulnerabilities. Further on, a browser extension is typically installed for a long time and thus maintaining the attack is easy. Some browsers even propose the user to synchronize extensions and their data over multiple devices, which makes it even harder for a user to detect an attack. Maybe the biggest hurdle for a browser extension attack is to be accepted by the browser's extension store. However, multiple extensions have already successfully bypassed the checks (Brinkmann 2018; Goodin 2018).

Is this as frightening as is sounds? As a browser extension can, at least in Firefox, also steal any cookie regardless of the *Secure* and *HttpOnly* flags (cf. section A.3), the threat of a browser extension intercepting FIDO2 messages becomes relative. Never the less, it is important to realize the security limits of FIDO2. According to Grant et al., FIDO2 qualifies for the NIST's authenticator assurance level 3 (2017). However, FIDO2 is p.ex. not suited as a replacement for CrontoSign to authenticate a payment. In contrast to an out-of-band authenticator, a malicious browser extension could approve an abusive payment without user interaction. In order to prevent false security assumptions, this should be stated in the FIDO2 standards.

The cost of mounting an attack mainly depends on two factors: First, how difficult it is to get the rogue extension into the store (or make the victim install it as unofficial extension), second, the attestation type the relying party requires. As long, as the attestation types *none* (which is the standard's default) or *self* are allowed, it will only take an attacker a couple of days to craft the extension. For the other attestation types, the attacker must also setup and maintain the accomplice server, and – probably the most difficult part – have a matching genuine authenticator model plugged into the accomplice server combined with an automated mechanism to activate it. But, with some mechanical engineering skills, this should as well be feasible in a few days. Over all, if a bank would rely on FIDO2, I would classify the present attack as an interesting business case.

Would token binding solve the issue? It depends on the exact implementation. It only mitigates the attack, if a browser extension can not use the same TLS session as the browser itself. However, as the Chromium project abandoned token binding (Harper 2018) and Firefox never implemented it, the impact of token binding wasn't scrutinized.

Another possible mitigation would be to disallow any browser extensions for a secure browsing environment. This could be implemented as a special browsing mode like for private browsing, or the browser could provide an API for any website to temporarily disable browser extensions after user consent. A less intrusive method would be, to at least cease access to request and response data for any browser extension. All of those propositions must however be evaluated carefully.

As further research it would be interesting to apply the same attack to different browsers. Does it work in Chrome? How about browsers on mobile devices? Do FIDO2 requests of mobile apps use the browser stack? If yes, are extensions enabled? May thus a malicious extension even break the authentication in a native app?

In addition, it would be interesting to find out, how difficult it is to get a malicious browser extension into the official webstores. How about getting it in directly? And as an update to an existing extension? How diverse are the checks of the different vendors?

In conclusion I'd like to point out that, despite FIDO2's vulnerability for man-in-the-browser attacks, it must still be judged to provide considerably better security than traditional authentication with username and password. Most of FIDO2's security assumptions and goals do still uphold and they are measurably stronger than the security provided by username with password. However I would not recommend to rely on FIDO2, without an additional out-of-band verification, for high security applications.

References

Balfanz, Dirk et al. (Mar. 2019). *Web Authentication: An API for accessing Public Key Credentials, Level 1.* W3C. URL: https://www.w3.org/TR/2019/REC-webauthn-1-20190304/ (visited on 05/09/2020).

Brand, Christiaan et al. (Jan. 2019). *Client to Authenticator Protocol (CTAP).* FIDO Alliance. URL: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html (visited on 05/09/2020).

Brinkmann, Martin (May 2018). *Google's bad track record of malicious Chrome extensions continues.* Website. URL: https://www.ghacks.net/2018/05/11/googles-bad-track-record-of-malicious-chrome-extensions-continues/ (visited on 05/09/2020).

FIDO Alliance (2020). *FIDO Alliance - Open Authentication Standards More Secure than Passwords.* Website. URL: https://fidoalliance.org/ (visited on 05/08/2020).

Goodin, Dan (Jan. 2018). *Google Chrome extensions with 500,000 downloads found to be malicious.* Website. URL: https://arstechnica.com/information-technology/2018/01/500000-chrome-users-fall-prey-to-malicious-extensions-in-google-web-store/ (visited on 05/09/2020).

Grant, Jeremy et al. (2017). *NIST 800-63 Guidance & FIDO Authentication.* Webinar. URL: https://fidoalliance.org/event/webinar-nist-800-63-guidance-fido-authentication/ (visited on 05/31/2020).

Grassi, Paul et al. (2017). *Digital Identity Guidelines.* NIST Special Publication 800-63B. URL: https://pages.nist.gov/800-63-3/sp800-63b.html (visited on 06/01/2020).

Harper, Nick (Aug. 2018). *Intent to Remove: Token Binding.* blink-dev. URL: https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/OkdLUyYmY1E/w2ESAeshBgAJ.

Lindemann, Rolf et al. (Apr. 2017). *FIDO Security Reference.* FIDO Alliance. URL: https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-security-ref-v1.2-ps-20170411.html (visited on 05/09/2020).

MDN contributors (2020). *Browser Extensions.* URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions (visited on 05/30/2020).

statscounter (2020a). *Desktop Browser Market Share Worldwide.* Website. URL: https://gs.statcounter.com/browser-market-share/desktop/worldwide (visited on 05/09/2020).

— (2020b). *Mobile Browser Market Share Worldwide.* Website. URL: https://gs.statcounter.com/browser-market-share/mobile/worldwide (visited on 05/09/2020).

Stötzner, Miles (2018). "Über die (Un-)Sicherheit des W3C-WebAuthentication-Entwurfs: Eine Beschreibung und Sicherheitsanalyse". MA thesis. University of Stuttgart. URL: http://elib.uni-stuttgart.de/handle/11682/10389.

Stötzner, Miles et al. (2018). *Leap of Faith not only for Self and None Attestation Types.* Github Issue. URL: https://github.com/w3c/webauthn/issues/1088 (visited on 05/30/2020).

Glossary

ajax Methodology to issue ascynchronous HTTP requests in JavaScript. 12

authenticator The cryptographic device, that generates the credentials key pair and performs all actions that require the private key. 4

client In the context of FIDO2, the client usually refers to the browser. 4

CrontoSign App for e-banking access and transaction authentication. 16

FIDO Alliance Industry association behind FIDO2. 4, 22

P-256 Specific named curve for elliptic curve operations. 15

relying party The party that wants an authenticated user. Usually the server. 4

SHA-256 Standard hashing algorithm with 256 bit output value. 15

webpack JavaScript module bundler. 12

Acronyms

CAPT Client to Authenticator Protocol. 3, 4

DOM Document Object Model. 14

ECDSA Elliptic Curve Digital Signing Algorithm. 15

NIST National Institute of Standards and Technology. 3, 16

TLS Transport Layer Security. 16

TPM Trusted Platform Module. 4

W3C World Wide Web Consortium. 4

WebAuthN Web Authentication Protocol. 3, 4, 7, 8, 11, 12, 16

XSS Cross Site Scripting. 16

A. Annex

A.1 Security Assumptions

SA-1 The Authenticator and its cryptographic algorithms and parameters (key size, mode, output length, etc.) in use are not subject to unknown weaknesses that make them unfit for their purpose in encrypting, digitally signing, and authenticating messages.

SA-2 Operating system privilege separation mechanisms relied up on by the software modules involved in a FIDO operation on the user device perform as advertised. E.g. boundaries between user and kernel mode, between user accounts, and between applications (where applicable) are securely enforced and security principals can be mutually, securely identifiable.

SA-3 Applications on the user device are able to establish secure channels that provide trustworthy server authentication, and confidentiality and integrity for messages (e.g., through TLS).

SA-4 The computing environment on the FIDO user device and the and applications involved in a FIDO operation act as trustworthy agents of the user.

SA-5 The inherent value of a cryptographic key resides in the confidence it imparts, and this commodity decays with the passage of time, irrespective of any compromise event. As a result the effective assurance level of authenticators will be reduced over time.

SA-6 The computing resources at the Relying Party involved in processing a FIDO operation act as trustworthy agents of the Relying Party.

(Lindemann et al. 2017, sec. 6)

A.2 Security Goals

SG-1 Strong User Authentication: Authenticate (i.e. recognize) a user and/or a device to a relying party with high (cryptographic) strength.

SG-2 Credential Guessing Resilience: Provide robust protection against eavesdroppers, e.g. be resilient to physical observation, resilient to targeted impersonation, resilient to throttled and unthrottled guessing.

SG-3 Credential Disclosure Resilience: Be resilient to phishing attacks and real-time phishing attack, including resilience to online attacks by adversaries able to actively manipulate network traffic.

SG-4 Unlinkablity: Protect the protocol conversation such that any two relying parties cannot link the conversation to one user (i.e. be unlinkable).

SG-5 Verifier Leak Resilience: Be resilient to leaks from other relying parties. I.e., nothing that a verifier could possibly leak can help an attacker impersonate the user to another relying party.

SG-6 Authenticator Leak Resilience: Be resilient to leaks from other FIDO Authenticators. I.e., nothing that a particular FIDO Authenticator could possibly leak can help an attacker to impersonate any other user to any relying party.

SG-7 User Consent: Notify the user before a relationship to a new relying party is being established (requiring explicit consent).

SG-8 Limited PII: Limit the amount of personal identifiable information (PII) exposed to the relying party to the absolute minimum.

SG-9 Attestable Properties: Relying Party must be able to verify FIDO Authenticator model/-type (in order to calculate the associated risk).

SG-10 DoS Resistance: Be resilient to Denial of Service Attacks. I.e. prevent attackers from inserting invalid registration information for a legitimate user for the next login phase. Afterward, the legitimate user will not be able to login successfully anymore.

SG-11 Forgery Resistance: Be resilient to Forgery Attacks (Impersonation Attacks). I.e. prevent attackers from attempting to modify intercepted communications in order to masquerade as the legitimate user and login to the system.

SG-12 Parallel Session Resistance: Be resilient to Parallel Session Attacks. Without knowing a user's authentication credential, an attacker can masquerade as the legitimate user by creating a valid authentication message out of some eavesdropped communication between the user and the server.

SG-13 Forwarding Resistance: Be resilient to Forwarding and Replay Attacks. Having intercepted previous communications, an attacker can impersonate the legal user to authenticate to the system. The attacker can replay or forward the intercepted messages.

SG-14 Transaction Non-Repudiation: Provide strong cryptographic non-repudiation for secure transactions.

SG-15 Respect for Operating Environment Security Boundaries: Ensure that registrations and private key material as a shared system resource is appropriately protected according to the operating environment privilege boundaries in place on the FIDO user device.

SG-16 Assessable Level of Security: Ensure that the design and implementation of the Authenticator allows for the testing laboratory / FIDO Alliance to assess the level of security provided by the Authenticator.

(Lindemann et al. 2017, sec. 4)

A.3 Stealing Secure HttpOnly Cookie

A browser extension can steal any cookie regardless of the *Secure* and *HttpOnly* flags. A simple demonstration example for a Firefox browser extension:

```
cookieSpotter = input => console.log(
  input.responseHeaders.filter(obj => obj.name === 'set-cookie')[0]?.
      value)

browser.webRequest.onHeadersReceived.addListener(
  cookieSpotter,
  {urls: ['<all_urls>']},
  ['responseHeaders']
);
```