Assumptions Kill: State of Webauthn in Electron

Jyotirmay Chauhan, Prof. Jason Polakis

December 9, 2024

Abstract

The proliferation of phishing and credential stealing in today's Web has spawned a whole industry of password managers and security extensions which aim to provide security against malicious actors. However, there is well-accepted mode of authentication which renders this risk moot- passwordless authentication. Originally built for 2FA and MFA scenarios, hardware security keys are increasingly replacing passwords in high risk situations. Major companies like Google and Amazon recommend the use of hardware keys to prevent account compromise. However, as the transition to passwordless technologies gains momentum, research needs to be conducted to validate the security and privacy claims of these technologies in all possible use-cases. This paper presents the first analysis of Webauthn, a passwordless authentication protocol for websites, in Electron applications. We conduct a thorough literature review and develop Electron implementations to understand FIDO2 authentication in this context. We identify a vulnerability that allows origin spoofing via the `protocol.handle()` and `net.fetch()` methods, enabling attacks such as relying party impersonation and rogue key injection. We demonstrate the feasibility of these attacks on Windows and discuss their implications. Our findings highlight the challenges of securing authentication in cross-platform frameworks like Electron and emphasize the need for careful consideration of security assumptions when adapting web technologies to desktop applications.

1 Introduction

Web Authentication (Webauthn)[15], a standard developed by the World Wide Web Consortium (W3C) and the FIDO Alliance, aims to provide a secure and convenient alternative to traditional password-based authentication on the web. Webauthn leverages public key cryptography and hardware authenticators to enable users to register and authenticate with online services in a more secure manner, mitigating risks associated with password reuse, phishing, and credential theft.

The standard is part of the FIDO2 family of protocol which all offers a competitive alternative to traditional mode of password based authentication. FIDO2 is the successor to the Universal 2nd Factor (U2F) protocols and builds upon the original CTAP Client-to-Authenticator Protocol (CTAP) to provide strong, phishing-resistant authentication. It has two children; **1)Webauthn** for passwordless authentication for websites using browsers and **2)CTAP2** counterpart for non-browser applications. FIDO has repeatedly been proven to secure [9] [11]

FIDO Alliance has a certification standard for authenticators which are based on the guarantees they provide during a Webauthn transaction. The **L2** and above are resistant to malware, and circuit level attacks due to the presence of TEE - Trusted Execution Environment. **L1 , L1+** fulfill all the basic requirement outline under Webauthn however they don't posses a TEE. Each higher standard build upon the guarantees of the previous standard.

Webauthn has several advantage over traditional passwords. They can't be read and stolen unless the hardware authenticators itself have been stolen. Even if an attacker has physical access to the authentica-

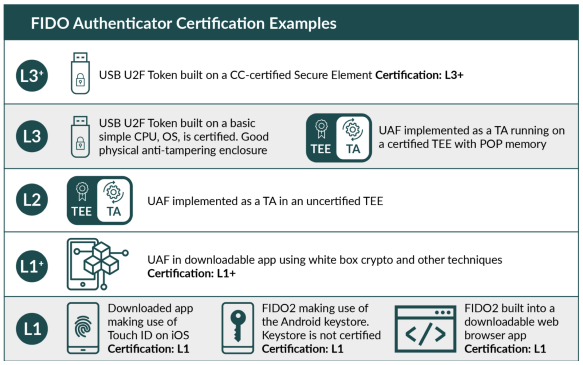

Figure 1: FIDO Certification Standard [8]

tor itself, they can't physically read the password (L2 and above); they can only extract a signed challenge assuming that the authenticator doesn't have any other safety features (e.g. master PIN) . Additionally, since the challenge being signed contains a unique nonce, an attacker can't collect signed challenges for future use. These advantages of hardware authenticator make them particularly hard to be compromised without gross negligence on the part of the user.

While the FIDO2 suite of protocol itself has been repeated been proven secure, problem often arise in its implementation in various environments. This project investigates the current state of FIDO2 authentication in Electron applications and identifies potential security vulnerabilities and attack vectors unique to the Electron ecosystem.

In this paper, we present the first analysis of Webauthn in Electron applications. In order to gain a comprehensive understanding of FIDO2 authentication in Electron applications, we conduct a thorough literature review of existing research on FIDO2 security focusing on web browsers. We also develop and test Electron implementations using a locally developed app and a corresponding website Relying Party (RP) supporting FIDO2 authentication . Building upon the insights gained from the literature review and the toy application, we also enumerate and test potential attacks against Webauthn in Electron. The main contribution of our research is as follows :

- We present a first analysis of the state of Webauthn in Electron Apps
- We perform a comprehensive literature review to build a repository of attacks against Webauthn
- We identify novel Electron-specific modules which allow us to perform an origin mismatch on a given web-page
- We show two new attacks against Webauthn using the origin spoofing to steal credentials

2 Background

2.1 Electron

Electron is a modern open-source framework that enables developers to build cross-platform desktop applications using web technologies such as JavaScript, HTML, and CSS. Fundamentally, Electron combines two key components: the Chromium rendering engine and the Node.js runtime. The Chromium rendering engine is responsible for displaying the application's user interface and handling web page rendering. Node.js, on the other hand, is a JavaScript runtime built on Chrome's V8 JavaScript engine, which allows developers to run JavaScript code outside of a web browser environment. Electron abstracts away many of the differences between operating systems, providing a consistent development experience and reducing the time and effort required to maintain separate codebases for each platform. Many well-known applications, such as Slack, Visual Studio Code, WhatsApp, and Figma, are built using Electron. Packing websites into Electron apps allows developers to create feature-rich desktop applications that can run on multiple operating systems, including Windows, macOS, and Linux, using a single codebase- a distinct advantage of complex apps

However, it is important to note that Electron applications come with certain trade-offs. Use of web technologies and the Chromium rendering engine can result in larger application bundle sizes compared to native applications. Additionally their performance may not match that of native applications in certain

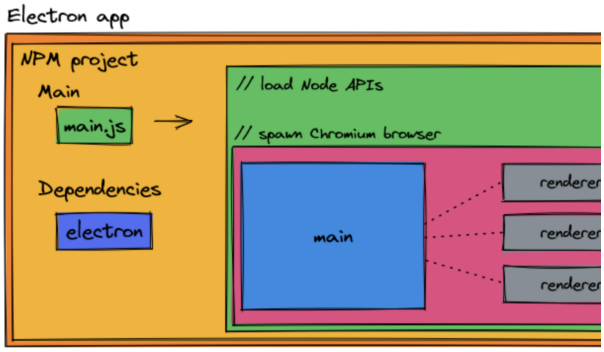

Figure 2: Chromium Architecture [16]

scenarios. Most importantly, while Electron apps behave like browsers, they are neither browsers nor are they typical web apps.

Security: Electron unique architecture also introduces unique security challenges. As platform based applications Electron applications have significantly elevated privileges compared to traditional web-app running in a browser environment. But because of the same underlying technologies, Electron apps can suffer from similar security issues such as XSS attacks but with potentially more severe consequences due to the increased privileges such as getting remote code execution.

Architecture: The architecture of Electron matches that of a typical web browser. Electron applications consist of two main processes: the main process and the renderer process. The main process, which runs on Node.js, is responsible for creating and managing application windows, handling system-level events, and interacting with the operating system's native APIs. The renderer process, which runs on Chromium, is responsible for rendering the application's user interface and executing the JavaScript code within the app logic. The separation of Main and renderer processes and the sandboxing of the renderer process are the status quo for all modern web browsers. Electron exemplifies this distinction in it APIs as well. The Main process API have access to system resources and also to the Node environment.

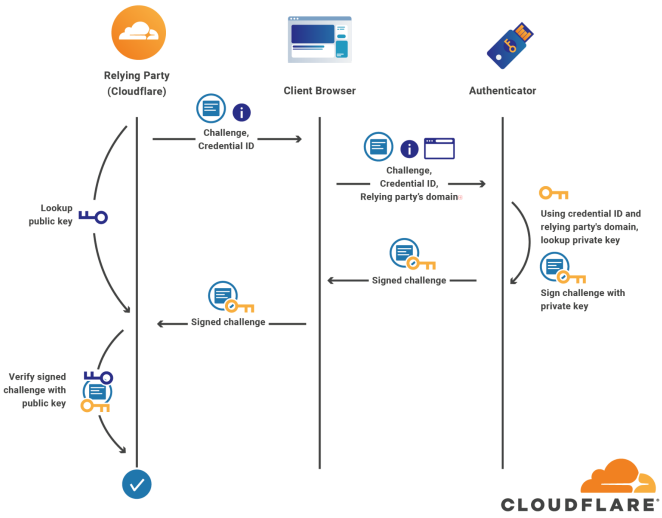

Figure 3: Webauthn Authentication Handshake [6]

On the other hand, typically, the renderer process can't access the Node API since it loads untrusted content.

2.2 Webauthn

Webauthn is part of the FIDO2, the second generation of protocols for passwordless-authentication. It is security protocol designed to transmit authenticity without revealing the secret which establishes it. For a proper understanding of the guarantees provided by Webauthn, some definitions are essential. We will use *Relying Party*, to denote a organization/website/portal who wants to ascertain an user's credentials. Any typical website which requires logging in fits in this description. *User* will be used to denote an actor or an organization who posses and owns the hardware authenticator and is trying to confirm their identity to the relying party.

In the protocol itself, the following entities are involved:

- **Authenticator** refers to the hardware key (such as Yubikey) or software authenticator (such as

Google Authenticator). It is the device which stores the private keys and contains the TEE-Trusted Execution Environment.

- **ASM** stands for Authenticator Specific Module and provides a standardized software interface to the Authenticator
- **FIDO Client** is the software which sits between the user and the Authenticator and facilitates interaction.
- **FIDO Server** is the server maintained by the Relying-Party backend to store the user information in a webauthn transaction. It stores both the User identification as well as their corresponding public key. This is absolutely necessary in every transaction

The transmission of trust occurs via the principle of public key cryptography[12]. While a full explanation of public key cryptography is outside the scope of this paper, but a basic explanation is as follows: Public key cryptography separates the encryption and the decryption keys. This unique feature of public key cryptography allows a party is able to encrypt a message without knowing a way to decrypt it and vice versa. This principle is used to trust websites, and match code signatures. In Webauthn, this principle is used to identify the user.

At the time of the registration, the hardware authenticator creates a pair of public and private keys. The public keys are used to encrypt a message and the private keys are used to decrypt it. The public keys are passed onto the Replying party. The private key never leaves the the hardware authenticator. Since any message encrypted by the public key can only be decrypted by the hardware authenticator's private key, the Relying Party holding the public key can authenticate the user. The Webauthn standard create such public-key private-key pairs for each individual relying party. The mechanism works as follows: -

2.2.1 Registration

1. The Website Sends a Registration Request as PublicKeyCredentialObject with the following options
    - **Relying Party (RP) information:** The website's domain name and other relevant details.
    - **User information:** A user handle or user ID to associate with the registration.
    - **Challenge:** A randomly generated byte array used to ensure the registration request is unique and prevent replay attacks.
    - **Authenticator selection criteria:** Specifies the types of authenticators allowed (e.g., platform authenticators, roaming authenticators).
2. The website sends the registration request to the user's browser. The browser receives the registration request and prompts the user to select an authenticator (if multiple authenticators are available). The selected authenticator generates a new cryptographic key pair consisting of a private key and a public key. The private key is securely stored on the authenticator and never leaves the device. The public key is returned to the browser. The authenticator creates an attestation object, which includes an attestation statement that provides information about the authenticator, such as its type, model-ID *(AAGUID)*, and certification. The certificates chain back to a trusted root certificate demonstrating the provenance of the authenticator.
3. Browser sends back registration response back to the Relying Party. The Relying Party verifies the attestation object by checking the signature and ensuring the authenticator meets the required criteria and extracts the public key from the attestation object and associates it with the user's account.

The registration is now complete, and the public key is stored for future authentication. The RP can verifiably determine the legitimacy of the authenticator and any associated risks [13]

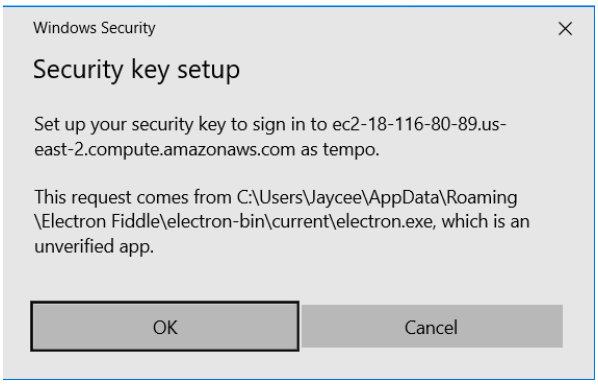

Figure 4: Popup on Windows

2.2.2 Authentication

After the registration process, the user can authenticate using the same authenticator. The handshake once again uses a PublicKeyCredentialObject and occurs in a similar manner.

1. User Initiates Authentication; usually by clicking on the Sign In button
2. The Relying Party prepares and sends an authentication request, which includes the following parameters:
   - Relying Party (RP) information
   - Challenge
   - Allowed credentials
3. The browser receives the authentication request and checks if any of the allowed credentials are available on the user's authenticators. If a matching credential is found, the browser prompts the user to select the authenticator (if multiple authenticators are available).
4. The selected authenticator receives the authentication request. It retrieves the private key associated with the credential ID specified in the allowed credentials and uses the private key to sign the authentication request, creating an authentication assertion. The authenticator may require user verification (e.g., PIN, biometric, or touch) before signing the request.
5. The authenticator returns the authentication assertion to the browser.

   which in turn collects the authentication assertion and passes it to the website. The website verifies the authentication assertion by checking the signature using the public key associated with the credential ID. If the verification succeeds, the website considers the user authenticated.

3 Literature Review

FIDO's design of the Webauthn protocol comes with certain guarantees. For example, the protocol guarantees that the FIDO credential can only be accessed by the original user associated with the credential. The authenticators are expected to enforce this using local authentication (e.g. PIN or Fingerprint). The strength of this enforcement has in impact o their security rating as well. The protocol also guarantees that only the same client application and ASM which was used during registration can be used during authentication.[7]. These guarantees ensure that certain attacks are not possible:

Replay Attacks are attacks, in which a malicious actor uses a prior signed challenge to attempt to gain access to the RP. Webauthn defeats this using the challenge nonce. Since an old signed challenge will contain an old challenge nonce, the RP upon decrypting the signed challenge can reject a replay. The standard explicitly states that each challenge sent by the RP must have new random nonce.

Phishing Attacks also fail for the same reason. Since at best, an attacker can obtain is signed challenge, passive phishing attempts are fruitless as a signed challenge cannot be reused. Additionally, phishing page rely on social engineering to fool the victims into believing that they are on the right website. However, in order to fool an authenticator, the attacker need to spoof the phishing page's origin, which is a very difficult.

The Webauthn protocol also implements a *KHAccessToken* which binds a credential with its private key. It is generated using the following:

- AppID - identifier of an application which is being used to access the credential.
- PersonalID - random identifier used to differentiate between different credentials with the same RP.
- ASMToken - ASM identifier hardcoded into the key at creation
- CallerID - ID of the application/website attempting to retrive a signed challenge

To build a KHAccessToken with matches a particular credential, all these four elements need to match. This prevents a user-level malware from stealing credentials as it can't spoof the above values without root privileges. [13]. Thus, FIDO2 protocols are generally considered secure if properly implemented.

Prior Work: There has been considerable attempts by various researchers to launch an attack on FIDO ecosystem. Barbosa et al [10] proposed two attacks: Rogue key and impersonation attacks. In Rogue key attacks, malware replaces an the victim's key is with the attacker key whereas in Impersonation attacks, the attacker Man-in-the-Middles between the user and the RP[10]. Other paper have looked at Double binding attacks[14] where a attacker preemptively binds a hardware token with the users Relying Party as well as timing based attacks such as synchronized login and user ID detection using side channel. There have also been social engineering attacks to exploit MFA fatigue.[13]

However all these attacks have a common theme: they require a strong attacker position. All of the attacks listed above require either a malicious browser extension with scripting privileges or at-least a user level malware.

4 Design

4.1 Vulnerablity

The main focus of the paper is on spoofing the RP's credentials being sent by the Electron app to hardware authenticator. We know that the Authenticator registers each protocol using the origin of the website of the RP. If this origin can be spoofed, then for all intents and purposes, the authenticator can't differentiate between the malicious actor and the RP.

Origin spoofing is possible in Electron app via the `protocol.handle()`[5] and the `net.fetch()` methods. These methods are Main process methods in Electron. The `protocol.handle()` is used to intercept all request origination form the app. The method takes in a schema an intercepts all outgoing requests using this particular scheme e.g. *'https'* , *'file'*. The `net.fetch()` method is a Electron own implementation of the browser fetch method which uses Chromium native networking API[4]. It is important to note here that Electron developers recommend its use over a traditional Node API fetch.

During our experiments, we noted that the whenever the Main process loads a URL using the `.loadURL()`, instance method of Browser Window, the app sets the origin of this page to that of the URL. However if this request is intercepted via `protocol.handle` and changed to another URL or URI, the app doesn't change the page origin to match the new URL. This bug is persistent and present across all platform (Windows, Linux, macOS) Using other fetch methods doesn't achieve the same results.

The following attacker models utilize this vulnerability to steal FIDO credentials via phishing.

4.2 Threat Model

We assume that the attacker doesn't have physical access to the hardware authenticator or the RP backend. In line with FIDO protocol assumptions, We also assume that the attacker has not compromised either of them prior to the launch of the attack.

We present the following two models for origin spoofing in Electron apps.

4.2.1 Malicious Electron App

Under this model, the user/victim has installed a malicious Electron app which used as an intermediary in the Webauthn transaction. The likely situation can be an app which categorizes user's Reddit feed and allows the user to browser Reddit posts and open any enclosed links. The app must have the ability to

open web-page in app using Electron's built-in API such as BrowserWindow, Webview etc. The attack is possible when the user is using this application for a Webauthn transaction. The authors recognize that this is the strong assumption of the attack.

When the user navigates to a target URL, the app load the URL in a new browser window using the instance method. However before the page can be loaded, the request is intercepted by Electron's `protocol.handle()` which is loads a local phishing page instead. The phishing page matches the page originally being loaded by `.loadURL()` so the user can't tell the difference.

When a user attempts a FIDO transaction and the browser sends the PublicKeyObject to the authenticator, the RP details included in the object match those of the original `.loadURL()` even though the page is not being sent by that origin.

This allows the attacker to arbitrarily decide the origin in any request to the hardware authenticator. However since the popup shows the origin of the request initiator to the user, the attacker can stealthily spoof only the intercepted URL's origin.

In essence, the attacker can now be in the middle of all FIDO transactions between the RP and the user. In this position, the attacker can deploy various attacks e.g. Relying party Impersonation or Rogue key attacks [10] and compromise user credentials.

4.2.2 Improper Sanitization via IPC

Inter-Process Communication (IPC) in Electron enables communication between the Main process and the Renderer processes. While the renderer process are sand-boxed[1] from the main process, they still need a method to communicate with each other . This communication is essential for coordinating actions, sharing data, and responding to events across different parts of the application. Electron has the following built-in modules for IPC messaging *IPC Main* [2] for the Main module and *IPC Renderer* [3] for all renderer processes. A typical usecase for IPC messaging can be a renderer process sending a URL link to the Main process to open the link into a new window. (A properly sand-boxed and context-isolated renderer process doesn't have the privilege to do this)

IPC security: Electron documentation provides several security guidelines about how to properly implement and secure an IPC channel. One of the measures include is proper sanitization of any user input value before sending it to the Main process. Prior research into Electron has found out this is often not done correctly. We assume that the vulnerable Electron app being targeted has the same flaw.

Threat Model: We assume that the user is performing a Webauthn transaction using a vulnerable Electron app. The vulnerable Electron app has improper input sanitization and uses `protocol.handle( )` and `net.fetch( )` to open new browser windows based on input from the user.

The user is made to click on malicious URL which opens an attacker controlled phishing website in a renderer process. This page requires a user to Sign In via the target RP. When the user clicks on the fake sign-in link, `protocol.handle( )` intercepts the request a replace it with an attacker controlled URL which is then returned via `net.fetch( )`. The replaced URL leads to phishing websites which mirror the actual RP's page. In our testing we found out that `net.fetch( )` can be used to fetch another third party website without changing the origin. So once again we can spoof the origin and serve a phishing page; however, this page doesn't have to be local. The attacks discussed before, Rogue key attacks and Impersonation attacks, can now be conducted.

5 Evaluation

We hosted multiple EC2 instances on difference origins to function as the attacker-controlled phishing website and a relying party. We also created 2 kinds of Electron app: 1) Malicous App 2) Vulnerable App. For the hardware authenticator, we used a Yubikey 5C. We ran this experiment in various OS environments but were only able to conduct the experiment successfully in Windows. Experiments on macOS and Linux were unable to get an Electron app to open a user popup for the Webauthn transaction. Browsers (Chrome,Firefox,Safari) however were able to do this successfully.

On Windows, We attempted several registration and authentication ceremonies using a combination of these tools to register as user on against various domains. We were able to register fake keys on the Authenticator for prominent domains such as *'google.com','github.com'* on the hardware authenticator. We were also able to extract signed FIDO challenges from the authenticator for any origin.

5.1 App Evaluation

We also took a deep dive into Github Desktop, which support Webauthn authentication protocol, to identify the authentication process. We found that while the app uses the `protocol` module for intercepting certain schemes, it doesn't support in app Webauthn authentication. Instead, the application utilized deep links to conduct authentication and registration transactions in the default browser.

We also took a look at the Agora Flat app which is built using Electron. While it doesn't support Webauthn however, it uses `protocol.registerFileProtocol`. However, we were unable to find a working attack.

6 Discussion

The origin mismatch vulnerability is serious one because several important security features depend on origins. If the origin of a web-page can be called into question, various attacks may be possible including

1. **Cross-Site Scripting (XSS):** Inject malicious scripts into the application, stealing sensitive data or performing unauthorized actions.

2. **Cross-Site Request Forgery (CSRF):** Trick the application into making unintended requests to external websites, potentially compromising user data or performing actions on the user's behalf.

3. **Privilege Escalation:** Gain unauthorized access to privileged APIs or resources by impersonating a trusted origin.

4. **Data Tampering:** Modify or manipulate data sent between the application and the intended origin, leading to data corruption or unauthorized modifications.

The crux of the origin mismatch vulnerability lies in the interplay of `protocol.handle()` and `net.fetch()`. While further examination of Electron source code is required to understand the true origin of the problem, the bug's existence has simpler explanation.

Electron doesn't fit in well under either categorization outlined by the FIDO Alliance: They are neither a completely native app, nor are they web browsers. This presents an interesting dilemma: Should a credential be tied the code signature of Electron app (CTAP) or origin of the website/RP requesting the credential (Webauthn). As we have seen, Electron ties credentials to the RP which makes the above attacks possible.

It is relevant to mention that our first attacker model matches the attacker position of attacks detailed by prior research. A malicious Electron app is a strong attacker position. However, our second attacker model is not quite as strong. the attacker doesn't need to seed malware for the exploit to work. However, the conditions under which the attack is made possible are more demanding when compared to the previous scenario.

6.1 Future Work

Further work needs to be done on `protocol.handle()` and `net.fetch()`. Since these modules also used in other scenarios (e.g. intercepting a '*file://*' URI scheme and opening the local file in app) , this vulnerability can have a outsized impact beyond Webauthn. The next step in this direction is analysing the Electron source code to pinpoint the origination of the bug.

Further analysis of GitHub Desktop's authentication process can be quite helpful as well. This can also be done by looking at its code-base.

7 Conclusion

Cross-platform frameworks like Electron are gaining mainstream appeal as they allow developers to reuse existing code to provide a similar user experience on a new platform. However assumptions about security and privacy may not be similarly transferable. The JavaScript landscape offers a plethora of different frameworks which can be used to achieve the same result; however, this doesn't mean that the similarly named modules and methods behave similarly. As our study reveals, such assumptions about security can lead to credential compromise in situations where this should be impossible. Our research shows that a native fetch implementation in Electron can be used to spoof a webpage's origin and compromise the Webauthn transaction. This aligns with prior work that has found that Electron framework can often regress user protections typically accepted as status quo in web security.

References

[1] Chromium docs - Sandbox.

[2] IPCMain — Electron.

[3] IPCRenderer — Electron.

[4] net — Electron.

[5] protocol — Electron.

[6] Cloudflare now supports security keys with Web Authentication (WebAuthn)!, 8 2020.

[7] FIDO APPID and FACET specification, 5 2022.

[8] F. Alliance. Certified Authenticator Levels - FIDO Alliance, 11 2023.

[9] M. Barbosa, A. Boldyreva, S. Chen, and B. Warinschi. Provable security analysis of fido2. In *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part III*, page 125–156, Berlin, Heidelberg, 2021. Springer-Verlag.

[10] M. Barbosa, A. Cirne, and L. Esquível. Rogue key and impersonation attacks on fido2: From theory to practice. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–11, 2023.

[11] N. Bindel, C. Cremers, and M. Zhao. Fido2, ctap 2.1, and webauthn 2: Provable security and post-quantum instantiation. Cryptology ePrint Archive, Paper 2022/1029, 2022. https://eprint.iacr.org/2022/1029.

[12] W. contributors. Public-key cryptography, 4 2024.

[13] D. Kuchhal, M. Saad, A. Oest, and F. Li. Evaluating the security posture of real-world fido2 deployments. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2381–2395, 2023.

[14] H. Li, X. Pan, X. Wang, H. Feng, and C. Shi. Authenticator rebinding attack of the uaf protocol on mobile devices. *Wireless Communications and Mobile Computing*, 2020:1–14, 2020.

[15] E. Lundberg, M. Jones, and A. Kumar. Web authentication: An API for accessing public key credentials - level 3. W3C working draft, W3C, Sept. 2023. https://www.w3.org/TR/2023/WD-webauthn-3-20230927/.

[16] reZach. The ultimate Electron guide, 11 2023.