Passwordless Authentication Using WebAuthn and OpenSK

Rajeev KARUVATH
Rochester Institute of Technology
Rochester, NY, USA
rk3824@rit.edu

Introduction

Background

Traditional passwords have been around since the nascent stage of computing and have been the de facto standard of authentication ever since. Although they have been the norm for such a long period, weak passwords have also been responsible for some of the major breaches in history. Passwords have directly or indirectly caused 81% of data breaches [6], with problems such as rampant password reuse severely compounding issues like weak passwords. Despite the meteoric rise of the Internet, an exponential increase in computing power availability & a significant decrease in manufacturing costs of computing devices, the common core of most authentication protocols still relies on the idea of a password, whose security is heavily dependent on the end user. [1]

There have been multiple attempts at mitigating this issue. Password managers are a classic example, where complex and compliant passwords are automatically generated & managed by the manager and the user is expected to remember merely the master password. However, even they are not immune to breaches, as evidenced by incidents at prominent service providers like LastPass [7]. Also, password managers suffer from major usability issues [8] that affect their mass adoption, and a majority of the general demographic still avoids them.

The core flaw of password-based authentication is the reliance on the average user to create a secure password (a standard that keeps changing with an increase in brute-forcing compute capabilities). Passwordless authentication aims to provide secure authentication without relying on any passwords at all [9], thereby eliminating the need for users to remember them in the first place. By leveraging advances in modern cryptography & employing performant compute power, passwordless authentication provides the user with a seamless experience while ensuring strong security.

W3C (World Wide Web Consortium), the de facto organization for Web standards, has proposed a passwordless authentication protocol called WebAuthn [4]. It takes advantage of public-key cryptography, and provides a standardized API (Application Programming Interface) for web browsers to interact with cryptographic devices (called “authenticators”) to securely perform PKI operations and perform registration (i.e enrollment) & login.

Such cryptographic devices, also popularly known as “Security Keys”, are becoming increasingly common these days. These devices are designed to be tamper-proof, and provide a cryptographically secure environment to perform operations like key generation, signature generation & verification and One Time Passcodes (OTP) generation [10]. Although they are currently used as part of Two Factor Authentication (2FA) protocols like FIDO U2F (Universal Two Factor), the security of such devices has been vetted to a standard to provide even single-factor authentication, similar to a Common Access Card (CAC) smartcard. [11]

Design

The implementation is based upon using the WebAuthn protocol to perform user enrollment & login using a dedicated security key. The overall design of this implementation project consists of two components :

1. Implementing a WebAuthn compatible authenticator using Google’s OpenSK framework
2. Implementing a web portal (with both client and server side code) that conforms to the WebAuthn protocol

This design will be tested by verifying the enrollment and login process on the browser for a particular username, across different device platforms (Windows, Android & iOS) and different browsers.

Implementation

WebAuthn Protocol

Registration

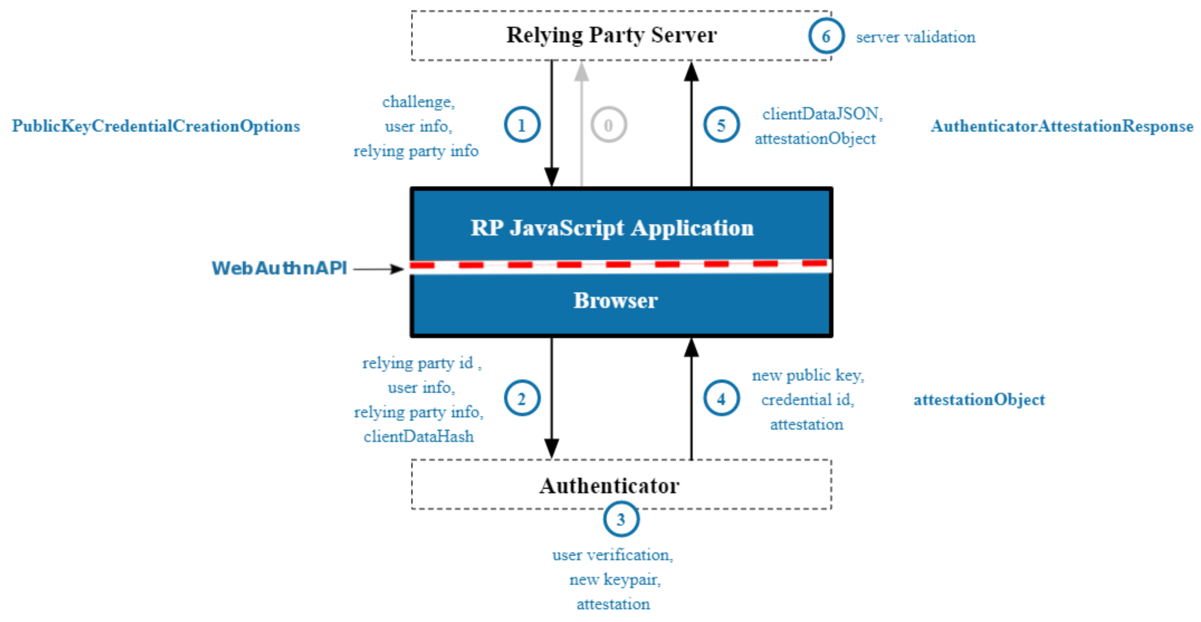


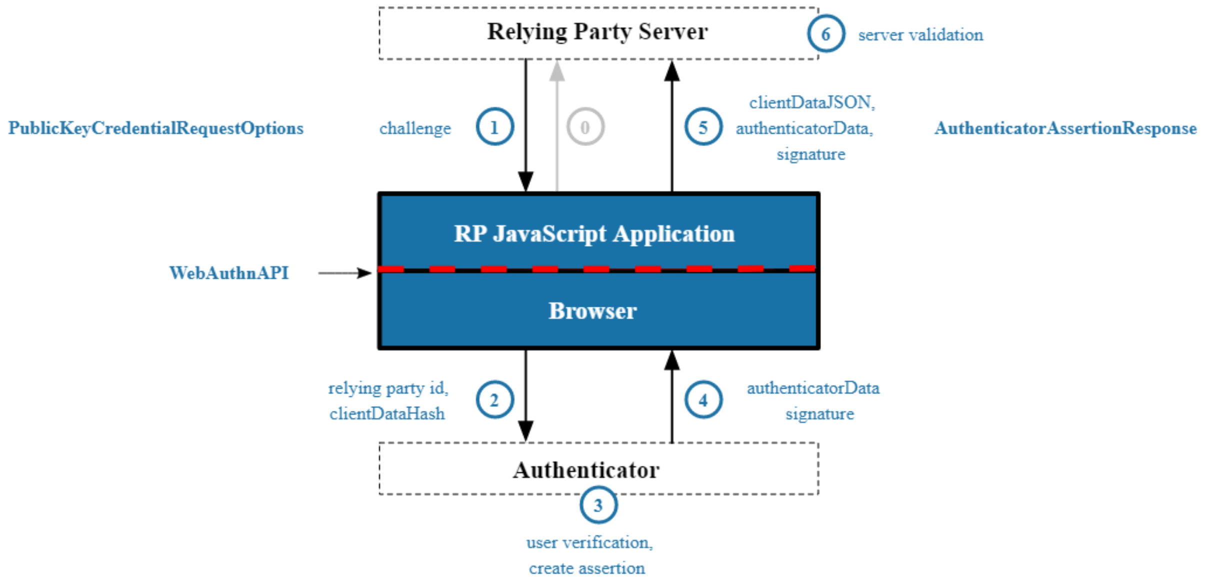
Figure 1 : WebAuthn Registration Ceremony [4]

A high level overview of Registration flow of WebAuthn is stated below :

1. The client (the browser in this case) first requests a challenge for a particular username
2. The server responds with a cryptographically secure challenge, and information about the website (also called "Relying Party")
3. The client sends this response to the "authenticator" (Security key in this case) via the CTAP2 protocol
4. Authenticator* generates a new key pair of credentials, and returns the public key and credential id to the client
5. The client formats this data and sends it to the server
6. The server verifies the challenge and the integrity of the data. If this is verified, registration is successful and the server can save the public key credentials.

* At this point, the authenticator can also request "user verification" from the client before proceeding. This user verification can be in the form of a PIN or biometric. .

Authentication



Relying Party Server
6
server validation
PublicKeyCredentialRequestOptions
challenge
1
0
5
clientDataJSON, authenticatorData, signature
AuthenticatorAssertionResponse
RP JavaScript Application
WebAuthnAPI
Browser
relying party id, clientDataHash
2
4
authenticatorData signature
Authenticator
3
user verification, create assertion

Figure 2 : WebAuthn Authentication Ceremony [4]

A high-level overview of Authentication flow of WebAuthn is stated below

1. The client (the browser in this case) first requests a challenge for a particular username
2. The server responds with a cryptographically secure challenge, and list of credential ids associated with the username
3. The client sends this response to the authenticator via CTAP2 protocol
4. Authenticator signs the response with the private key associated with the credential id, and generates a signature
5. The client formats this signature and sends it to the server
6. The server verifies the signature using the saved public key for that username. If verification is successful, the authentication flow is considered validated.

Architecture

Implementation of WebAuthn compliant security key

WebAuthn provides a standard framework for defining an “Authenticator”, i.e a cryptographic device capable of two main processes :

1. Generate PKI pairs (“Public Key Credential”) mapped to a particular web portal (or “Relying Party”)
2. Verifying a user by signing a challenge generated by the server

Many platforms conform with this standard, for e.g., the Windows Hello Platform, Apple's Secure Enclave [12], Android Security Key platform, etc. However, such platforms are not considered portable as they rely on cryptographic hardware specific to the host device, i.e., laptop or the smartphone itself.

For this implementation, the FIDO2 authenticator standard has been selected. FIDO (Fast Identity Online) is a group of 260 industry leaders like Google, RSA, Samsung, and more. FIDO2 is the successor to the original FIDO U2F standard, which allowed a security key to be used as a two factor token. With the FIDO2 [2] [5] standard, a security key can be used as a standalone single factor authentication device. The FIDO standards allow interaction with a FIDO compliant device through a protocol known as CTAP (Client to Authenticator Protocol) [3], which supports three major underlying "transports" : USB (wired), NFC (Wireless), BLE (Wireless). For this implementation, USB transport will be used.

The hardware platform selected is the Nordic nRF52840-dongle, and will be programmed with the FIDO2 compliant firmware using Google's OpenSK framework. The OpenSK framework simplifies the process of creating a Rust based TockOS application deployed on the Nordic dongle, which enables the dongle to connect to the host through the USB HID (Human Interface Device) standard, emulating a FIDO2 compliant security key.

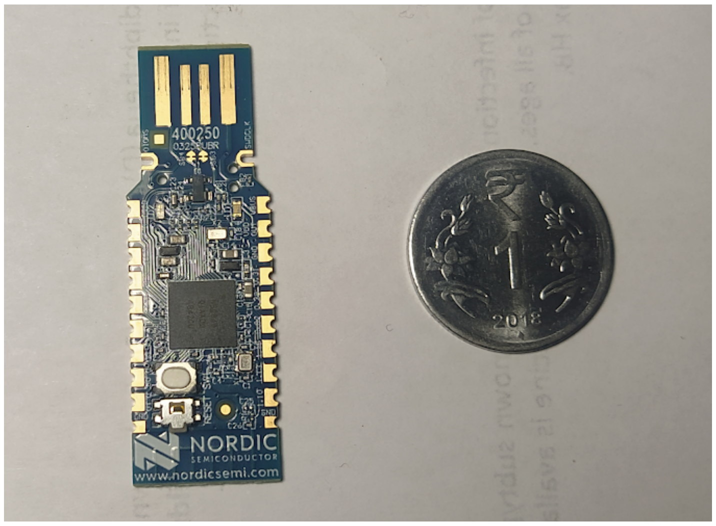


Figure 3 : Nordic nRF52840-dongle (1 ₹ coin for scale)

Implementation of WebAuthn compliant web portal

The web portal consists of two components :

Client Side Implementation

The client side of the web portal is the user facing User Interface, which has been implemented using the popular Angular 11 framework by Google. It is written in the strongly typed TypeScript language, leveraging the Angular Material library to show an aesthetically pleasing user interface.

The actual WebAuthn protocol is implemented by building a reusable Angular Service, which is injected into the main UI component and interacts with the server side WebAuthn implementation via a REST JSON (JavaScript Object Notation) API. Verbose details of the client side implementation are mentioned in the Description section.

The client side code & development environment setup instructions can be found at https://github.com/rajeevravindran/csec604termprojectwebauthn-frontend

Server Side Implementation

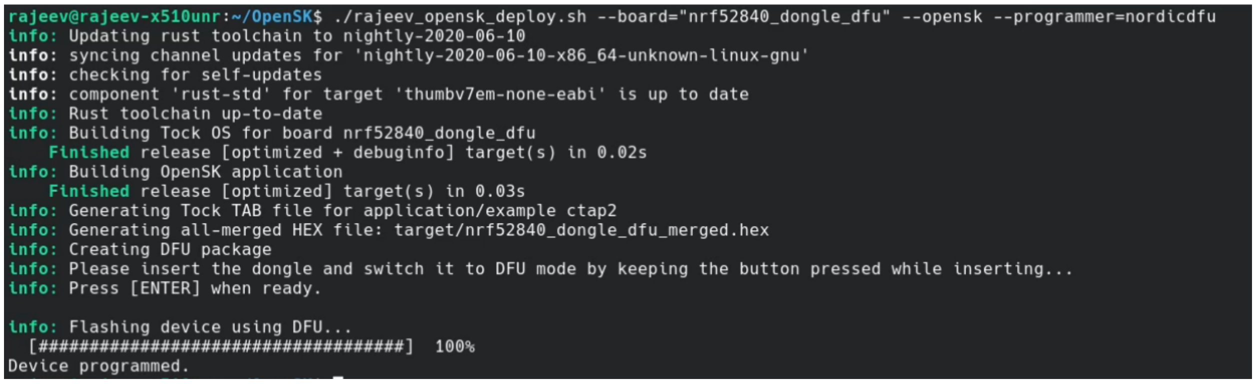
The server side of the web portal is the backend service that exposes WebAuthn functionalities like challenge generation, verification & user credential capture via HTTP REST JSON APIs . It has been implemented using the popular Python based web framework Django 3. User profile information, credentials, server side session management and other miscellaneous persistence is stored using sqlite3 as a relational database backend. Each exposed API is explained in detail in the Description section.

The server side code & development environment setup instructions can be found at https://github.com/rajeevravindran/csec604termprojectwebauthn

Description

OpenSK Firmware Flashing

First, the Nordic nRF52840-dongle was booted into DFU (Device Firmware Upgrade) mode, to allow for firmware upgrade. This was achieved by holding down the reset button while connecting the device to the host laptop. The host laptop runs Linux* Ubuntu 20.04 for maximum compatibility with the Rust compiler and miscellaneous Nordic toolchain required to interact with the device.
Next, the FIDO2 compliant TockOS firmware is compiled and flashed onto the device using a custom script written to automate some aspects of the build process.



```
rajeev@rajeev-x510unr:~/OpenSK$ ./rajeev_opensk_deploy.sh --board="nrf52840_dongle_dfu" --opensk --programmer=nordicdfu
info: Updating rust toolchain to nightly-2020-06-10
info: syncing channel updates for 'nightly-2020-06-10-x86_64-unknown-linux-gnu'
info: checking for self-updates
info: component 'rust-std' for target 'thumbv7em-none-eabi' is up to date
info: Rust toolchain up-to-date
info: Building Tock OS for board nrf52840_dongle_dfu
    Finished release [optimized + debuginfo] target(s) in 0.02s
info: Building OpenSK application
    Finished release [optimized] target(s) in 0.03s
info: Generating Tock TAB file for application/example ctap2
info: Generating all-merged HEX file: target/nrf52840_dongle_dfu_merged.hex
info: Creating DFU package
info: Please insert the dongle and switch it to DFU mode by keeping the button pressed while inserting...
info: Press [ENTER] when ready.

info: Flashing device using DFU...
  [####################################]  100%
Device programmed.
```

*Figure 4 : Firmware flashing process***

* The Linux host is required only for compatibility with the tools for compilation. Once the device is flashed with the firmware, the “Security Key” will be compatible with any modern OS capable of interacting with USB HID devices.
** The scripts and a video demonstration of the flashing process can be found in the Appendix.

Client-Side Implementation

The Angular client-side source code is divided into multiple components :

WebAuthnService

A reusable Angular service that can be injected into any component to make use of WebAuthn functionality.

```
startWebAuthnRegister(username: string, userDisplayName: string):
Observable<PublicKeyCredentialCreationOptions>
```

Before initiating the WebAuthn Registration ceremony, we first begin by requesting a cryptographic challenge from the server side API `/WebAuthn/register_WebAuthn_user` . The server returns the challenge parameters as base64 encoded, so the function converts these into ByteArrays so it can be used to start the Registration ceremony. The Registration Ceremony is initiated with `navigator.credentials.create({ publicKey: response })` where `response` is the server side challenge response procured earlier.

```
verifyWebAuthnRegister(signedCredentials: Credential | null):
Observable<any>
```

Once the user completes the ceremony (i.e., touches the security key button), `navigator.credentials.create` returns credentials signed by the user’s private key (embedded inside the security key), the public key and additional WebAuthn specific data as required by the specification. This data is converted from ByteArrays to base64 so it can be sent to the server-side API `/WebAuthn/verify_register_WebAuthn_user` for verification. If successful, it will open a dialog `RegisterSuccessComponent` with the registered credential details.

```
startWebAuthnLoginAttestation(username: string):
Observable<PublicKeyCredentialRequestOptions>
```

This function begins by first requesting a username specific challenge from the server-side API `/WebAuthn/login_WebAuthn_user_begin_assertion`. It also converts the challenge response parameters to ByteArrays. This converted challenge is passed to `navigator.credentials.get({ publicKey: response })` which initiates the Login Ceremony.

```
verifyWebAuthnLoginAttestation(loginAssertion: Credential | null):
Observable<any>
```

Once the Login ceremony is complete (i.e., the user touches the security key button), `navigator.credentials.get({ publicKey: response })` returns the signed credentials, similar to the registration flow. These signed credentials are converted to base64 and sent to server side API `/WebAuthn/login_WebAuthn_user_assertion_verify` for verification. If the server responds with a verified status, the dialog `LoginSuccessComponent` is opened.

AppComponent

The main bootstrap component which initiates the client side application. This component is responsible for rendering the UI. It consists of an Angular Material Form element allowing the user to enter “Username” and “Display Name”, buttons to trigger the Login or Register flow. The Login and Register buttons, when clicked, call the respective WebAuthnService functions to complete WebAuthn Authentication / Registration Ceremonies.

Server-Side Implementation

The server-side is implemented in Python Django, a very robust web framework that handles database management through an Object Relational Mapper (ORM), session management and much more.
For this implementation, the server side can be divided into four main REST API (corresponding to each client-side function)

`register_WebAuthn_user`

This function generates a challenge for the client-side to send to the authenticator. The challenge is generated using 32 random bytes created by a cryptographically secure pseudo random generator. On a Linux deployment, it calls **getrandom**() syscall which generates its entropy from *`/dev/urandom`* which provides access to environmental noise from device drivers etc. These random bytes are converted to base64 so it can be sent back to the client as an HTTP response. This function also saves the challenge in an anonymous session on the backend. The server also responds with a list of supported asymmetric key & hashing algorithms. This list conforms with the IANA CBOR Object Signing and Encryption standard. For e.g COSE_ALG_ES256 means Elliptic Curve Digital Signature Algorithm (ECDSA) with SHA-256 hashing.

`verify_register_WebAuthn_user`

This function verifies the response generated by the authenticator. We first retrieve the earlier challenge from the anonymous session. Although challenge details are encoded inside the authenticator response, this should not be trusted implicitly, but rather verified against the challenge saved in the session. In fact, this step verifies all the WebAuthn parameters like RP_ID, ORIGIN etc. This ensures the integrity of the challenge and miscellaneous WebAuthn parameters and generates the attestation signature using the verified details. If this attestation signature matches the decoded attestation signature, the ceremony is validated and the function then saves the decoded public key, credential id & username into the database as a `WebAuthnProfile` .

`login_WebAuthn_user_begin_assertion`

Similar to the start of the registration, this function generates a challenge. Then it extracts the given username's `WebAuthnProfile`. From the profile, the public key, credential id & signature counter are extracted. These details are formatted as `allowCredentials` for the user. This, along with the challenge, is sent back to the client.

`login_WebAuthn_user_assertion_verify`

Once the client authenticator signs the earlier sent assertion details, this function decodes the response to get the signed signature (which the client signed using the authenticator's embedded private key). This signature is then verified using the user's public key. If the verification is successful, non-repudiation is established and it is now confirmed the author of the signature possesses the private key associated with the given username's publickey, which was saved earlier during registration. The user is now logged in, and a success message is now returned.

Deployment

This project is deployed on https://csec-604-crypto-term-project.rajeevkr.me/ on a Google Cloud Platform server

The Angular artifacts were generated using the command `ng build --prod`

The Django backend was served on the server using Python Gunicorn, exposed on localhost:8000 and runs as a Systemd service. Systemd service code can be found in the Appendix

An Apache Virtualhost was configured to serve the Angular artifacts and reverse proxy backend API calls to the Gunicorn listener. A valid TLS certificate was obtained using LetsEncrypt certbot and deployed on the Apache server as well, for ensuring HTTPS compliance (mandated as part of WebAuthn spec). Apache Virtualhost conf file code can be found in the Appendix

Testing

Environment

For testing the setup, the following devices and browsers were used :

Device	OS Platform	Browser
Laptop	Windows 10	Chrome 86
Laptop	Windows 10	Firefox 83
Laptop	Windows 10	Edge 86
Laptop	Windows 10	Internet Explorer 11
Smartphone*	Android 10	Chrome Mobile
Tablet*	iPadOS 14	Safari Mobile

* For mobile devices, the Security Key was connected to the device using an USB-C OTG device

Methodology

Each device was tested on the website https://csec-604-crypto-term-project.rajeevkr.me/ through the following process

1. Total of five usernames were registered using the same security key
2. Registration time was recorded
3. Login was attempted using the five usernames
4. Login time was recorded

Due to the financial constraints of procuring additional security keys, a different methodology had to be adopted to emulate the failed authentication scenario. i.e., a user attempting to login for a username registered using a different security key.

To emulate this scenario, in the Django backend database, the user's saved public key was modified. A random character in the key was flipped to "emulate" a different key. This is used to validate failed signature verification and, in turn, rejected authentication.

Results

Device	OS Platform	Browser	Registration Time	Login Time

Laptop	Windows 10	Chrome 86	7.89	3.12
Laptop	Windows 10	Firefox 83	6.76	3.63
Laptop	Windows 10	Edge 86	7.12	2.98
Laptop	Windows 10	Internet Explorer 11	FAILED	FAILED
Smartphone	Android 10	Chrome Mobile	11.34	3.16
Tablet	iPadOS 14	Safari Mobile	FAILED	FAILED

All time values are in seconds, and are an average of each five username's ceremonies.

The registration and login flow worked seamlessly across all Windows browsers except Internet Explorer. This is because the WebAuthn specification is not supported by IE. IE is expected to be in the end-of-life phase now, with Microsoft pushing for Edge.

The registration flow took longer on the Android 10 smartphone. This could be because of three reasons :

1. A dropdown shows up asking the user for the preferred choice of the underlying transport, i.e., USB or Bluetooth based. This was not seen in the Windows platform.
2. USB OTG device takes some time to get initiated, and the Android permission framework kicks in asking user consent to access the device.
3. The Android permission framework kicks in again to ask consent to access the Security Key.

The WebAuthn flow did not work on the iPad tablet because it failed to detect the security key itself. This seems to be a compatibility issue with the USB C OTG cable.

Future Work

There is significant scope for future work on this project. The WebAuthn standard defines many underlying options that can be employed to further improve the security & privacy of a system.

TPM Attestation

WebAuthn allows for attestation using Trusted Platform Modules (TPM) found in most modern computing devices like smartphones and laptops. TPMs are isolated & dedicated cryptographic chips embedded in the device that can perform cryptographic operations like key-pair generation, secure storage of private keys & signing. They are also significantly faster thanks to the hardware implementation, and can resist side channel attacks due to the secure environment they operate in. Given this option, users do not need to buy separate security keys to take advantage of WebAuthn. Hence, the TPM attestation format can be implemented in the backend server side code to allow TPM attestation.

The only downside of using TPM attestation is that the credentials are not truly portable, since the private keys cannot be moved out of the device. This means an user enrolled using TPM attestation can login only from that device. However, this use case can be applicable in certain business use cases where the server wants to restrict access to resources only to specific hosts. For e.g., a corporate website designed to allow logins only from corporate laptops.

Account Lockout

Ideally, private keys should not be moved out of the security device. The WebAuthn specification strongly discourages backing up or copying private keys from security keys for preventing account lockouts. Rather, the specification allows for multiple credentials to be associated with a single user.

So, to prevent account lockout, the backend server side code could be enhanced to allow multiple credentials to be associated with a single user. The client side UI can be enhanced so that, after successful initial registration, a popup should request the user to register another security key with their account, to prevent account lockouts.

Privacy Improvements

The WebAuthn specification allows a wide gamut of options to minimize privacy concerns over the entire lifecycle of using WebAuthn. Options like Resident Key support can be implemented which allows passwordless login without a username. This addresses privacy concerns over database breaches where usernames are leaked, as usually usernames are not stored encrypted or hashed. The WebAuthn specifically also details out best practices that can be implemented, such as ensuring authentication ceremonies are initiated irrespective of the fact that the user does not exist, to prevent username enumeration attacks.

BLE Transport for Nordic nRF52840 Dongle

Nordic nRF52840 supports BLE (Bluetooth Low Energy) [13], so it is possible to modify the Google OpenSK framework to add support for BLE CTAP2 transport as well, to connect to the Security Key wirelessly. This will render the security key truly portable

Conclusion

The WebAuthn specification provides significant improvements over traditional authentication methods. On the client-side, it takes less effort to use a security key compared to entering passwords. Although there is an upfront investment required, i.e., purchasing a security key, it could definitely be worth it given the validated security of public key cryptography. Also, security keys could be a worthwhile investment for IT associated users as they can also be used for other tasks employing public key cryptography, like SSH logins & GPG encryption.

On the server side, there are significant security gains. Because no actual secret is transmitted to the server (either during registration or login phases), there will be no security fallout if the user database was breached in some way. The polar opposite is true for password breaches, where even salted hashed passwords can be cracked if they are sufficiently weak. Thanks to the “public” aspect of public key cryptography, this problem will never occur for WebAuthn based backend breaches.

Also, given the fact that WebAuthn supports multiple underlying transports like Bluetooth (BLE) & NFC, USB security keys do not have to be the sole mechanism for authentication. For e.g., Google’s Titan Security key supports Bluetooth.

The above reasons could serve as a strong motivation for eschewing traditional password based logins for such a passwordless authentication approach.

References

[1] Bonneau, J., Herley, C., Van Oorschot, P. C., & Stajano, F. (2012, May). The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy* (pp. 553-567). IEEE.

[2] Lyastani, S. G., Schilling, M., Neumayr, M., Backes, M., & Bugiel, S. (2020, May). Is FIDO2 the kingslayer of user authentication? A comparative usability study of FIDO2 passwordless authentication. In *2020 IEEE Symposium on Security and Privacy (SP)* (pp. 268-285). IEEE.

[3] Client to Authenticator Protocol (CTAP). fidoalliance.org. (2019) Retrieved 11 October 2020, from https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html

[4] Web Authentication: An API for accessing Public Key Credentials Level 1. W3.org. (2020). Retrieved 11 October 2020, from https://www.w3.org/TR/WebAuthn/.

[5] Chakraborty, D., & Bugiel, S. (2019, November). simFIDO: FIDO2 User Authentication with simTPM. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (pp. 2569-2571).

[6] Verizon Data Breach Report, verizondigitalmedia.com. (2017) Retrieved 11 October 2020, from https://www.verizondigitalmedia.com/blog/2017-verizon-data-breach-investigations-report/

[7] Whitney, L. (2011). LastPass CEO reveals details on security breach. *CNet, May*.

[8] Li, Z., He, W., Akhawe, D., & Song, D. (2014). The emperor’s new password manager: Security analysis of web-based password managers. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (pp. 465-479).

[9] Sælensminde, K., & Boonjing, V. (2010, April). A simple password less authentication system for web sites. In *2010 Seventh International Conference on Information Technology: New Generations* (pp. 132-137). IEEE.

[10] Lang, J., Czeskis, A., Balfanz, D., Schilder, M., & Srinivas, S. (2016, February). Security keys: Practical cryptographic second factors for the modern web. In International Conference on Financial Cryptography and Data Security (pp. 422-440). Springer, Berlin, Heidelberg.

[11] Alliance, S. C. (2016). Smart Card Technology and the FIDO Protocols.

[12] Mandt, T., Solnik, M., & Wang, D. (2016). Demystifying the secure enclave processor. Black Hat Las Vegas.

[13] Nordic Semiconductor, nordicsemi.com. (2020) Retrieved 11 Oct 2020 from https://www.nordicsemi.com/Software-and-Tools/Development-Kits/nRF52840-Dongle

Appendix

OpenSK Flashing Process - https://youtu.be/_3THvd6zSXs
WebAuthn Implementation Demo - https://youtu.be/0kWrMNat7Rg
WebAuthn Portability Demo - https://youtu.be/uXwuzuL5jos
WebAuthn Authentication Failure Demo - https://youtu.be/p1KE0DEh-As

Source Code

Client Side Angular Code
https://github.com/rajeevravindran/csec604termprojectwebauthn-frontend

Server Side Django Code
https://github.com/rajeevravindran/csec604termprojectwebauthn

Gunicorn Systemd Service
```
root@rajeev-portfolio:/home/rk3824# cat
/etc/systemd/system/csec604-gunicorn.service
[Unit]
Description=[PROD] csec-604-crypto-term-project.rajeevkr.me
After=network.target

[Service]
User=www-data
Group=www-data
WorkingDirectory=/home/rk3824/projects/csec604termprojectwebauthn
ExecStart=/home/rk3824/projects/csec604termprojectwebauthn-env/bin/gu
nicorn --workers 3 termprojectwebauthn.wsgi:application
```

```
[Install]
WantedBy=multi-user.target
```

Apache VirtualHost

```
root@rajeev-portfolio:/home/rk3824# cat
/etc/apache2/sites-available/csec-604-crypto-term-project.rajeevkr.me.conf

DEFINE domain_name csec-604-crypto-term-project.rajeevkr.me

<VirtualHost *:80>

        ServerName ${domain_name}
        DocumentRoot /var/www/${domain_name}
        Redirect permanent / https://${domain_name}

</VirtualHost>

<IfModule mod_ssl.c>
<VirtualHost *:443>

        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/${domain_name}
        ErrorLog ${APACHE_LOG_DIR}/${domain_name}.error.log
        CustomLog ${APACHE_LOG_DIR}/${domain_name}.access.log
combined

        ServerName ${domain_name}

<Directory /var/www/${domain_name}>
    AllowOverride All
</Directory>

ProxyPass /webauthn http://127.0.0.1:8000/webauthn
ProxyPassReverse /webauthn http://127.0.0.1:8000/webauthn

ProxyPass /admin http://127.0.0.1:8000/admin
ProxyPassReverse /admin http://127.0.0.1:8000/admin

Alias /static/
/home/rk3824/projects/csec604termprojectwebauthn/static/

<Directory /home/rk3824/projects/csec604termprojectwebauthn/static>
#       AllowOverride All
        Require all granted
</Directory>
```

```
ProxyPreserveHost on

Include /etc/letsencrypt/options-ssl-apache.conf
SSLCertificateFile /etc/letsencrypt/live/${domain_name}/fullchain.pem
SSLCertificateKeyFile
/etc/letsencrypt/live/${domain_name}/privkey.pem

</VirtualHost>
</IfModule>
```