Security Key Authentication for Web Applications

CS588 - Networked Distributed System Security | Professor J. Solworth

Davide Giacomini
dept. of Engineering
University of Illinois at Chicago (UIC)
Chicago (IL), USA
giacomini.davide@outlook.com

***Abstract*—The U2F (Universal 2nd Factor Authentication) protocol allows you to send a cryptographic challenge to a device (typically a key fob) owned by the user. A password starts the process, but the digital key is required to gain access. The FIDO U2F protocol was developed in 2014, and since then, the standards have been honed, refined, and updated. More users are growing accustomed to the idea of cryptographic keys. Some even demand this protection to keep their data safe and secure.**

In this brief report I will analyze the advantages of using a security key, and I will dive into the cryptographic protocol harnessed by those devices. Finally, I will present a sample web application, aimed at showing how current libraries and APIs can be exploited to ease the development of security key authentication in web applications.

***Index Terms*—web applications, authentication, security key, FIDO, U2F, security**

I. Introduction

In security, three main factors of authentication can be distinguished: *to know*, *to have* and *to be*. The first one requires the user to remember something, usually a password associated to the email or a username. The second one requires the user to own something during authentication, for example their own smartphone, or a physical key. The latter implies to recognize the user based on something that characterizes their uniqueness, for example from a fingerprint or their face.

Although passwords are inherently the weakest form of security to consider, they are actually the most widespread as form of user authentication on the web today [1]. Several are the techniques used to violate a password-protected account, among them there is phishing, guessing or spoofing [1]. Therefore, many have advocated for the use of a second factor for authenticating (2FA: 2 Factor Authentication) [2], thus limiting guessing and spoofing. Biometric authentication is becoming more and more widespread, especially with fingerprint scanners on phone, and in the last years with face scanner too. There are although several issues still not widely studied in biometric authentication, and systems still have defects that bring to their vulnerability. Bonneau et al. [2] surveyed these issues and also considered that some biological features have not been deeply studied yet.

Despite 2FA is extremely effective for better secure accounts, there are still attacks that can be carried out. For instance, OTPs can be victim of the so called real-time phishing attacks, which require the attacker to act as a "server-in-the-middle", pretending to be the real server and forwarding user's authentication requests and user's OTP to the real server [3]. However, message-based OTPs are the most used 2FA nowadays, especially because they rely on the assumption that a user will always carry their smartphone with them.

In this report, I will explain why security keys are theoretically better rather than other 2FA methods common today, bringing up problems in using other second factors and showing how they are avoided with a security key. I will then go through a sample web application that I developed in NodeJS[1] and VueJS[2], using Express[3] as framework, after looked at an overview of the cryptographic protocol used by Yubico[4] for their security keys.

II. Related Work

I would like now to report the most relevant related work in second factor authentication. Lang et al. [4] consulted a variety of excellent surveys work [2, 5, 6, 7] and listed five different technologies for 2FA, illustrating how security keys could fill some gaps in security left by those technologies. I will address four of them:

- **One-Time Passcodes:** Though OTPs provide more security than passwords, OTPs have a number of downsides. First, they are vulnerable to phishing and man-in-the-middle attacks, as I cited in Sec. I. Second, OTPs that are delivered by phones are subject to data and phone availability, while those that are generated by dongles cause the user to have one dongle per web site. Finally, OTPs provide a sub-optimal user experience as they often require the user to manually copy codes from one device to another. *Security Keys are resistant to phishing and*

[1] https://nodejs.org/en/
[2] https://vuejs.org/
[3] https://expressjs.com/
[4] https://yubico.com

man-in-the-middle by design; our preliminary study also shows that they provide a better user experience.

- **Smartphones as Second Factor:** While leveraging the users phone as a cryptographic second factor is promising, it faces a number of challenges: for example, protecting application logic from malware is difficult on a general purpose computing platform. Moreover, a users phone may not always be reachable: the phone may not have a data connection or the battery may have run out. *Security keys require no batteries and usually have a dedicated tamper-proof secure element.*
- **TLS Client Certificates:** Unfortunately, current implementations of TLS client certificates have a poor user experience. Typically, when web servers request that browsers generate a TLS client certificate, browsers display a dialog where the user must choose the certificate cipher and key lengtha cryptographic detail that is unfamiliar and confusing to most users. Accidentally choosing the wrong certificate will cause the users identity to leak across sites. TLS client certificates also suffer from a lack of portability: they are tough to move from one client platform to another. *Security Keys have none of these issues: they are designed to be simple to use, portable and fool-proof.*
- **Electronic National Identification Cards:** Some countries have deployed national electronic identification cards. Despite their rich capabilities, national identity cards require special hardware (a card reader) and thus are hard to deploy. Moreover, they are by definition controlled by one government, which may not be acceptable to businesses in another country and could arise a general concern about user's privacy. *Security Keys have no such downsides: they work with pre-installed drivers over commonly available physical media (USB, NFC, Bluetooth) and are not controlled or distributed by any single entity.*

III. Protocol Overview

I would like now to present an overview of the protocol used for the key I used to for myself. I used the Security Key NFC[5] by Yubico, which combines U2F and FIDO2 protocols [8]. For diving more into the protocol specifications, please consult the FIDO Alliance website[6].

The YubiKey is composed by an USB Interface and a Capacitive Touch Sensor (Fig. 1), useful for the Test of User Presence (TUP), which I will cover later.

The key supports to phases of the protocol: *register*, which is useful to register the key to new websites and, in turn, register the website as trustworthy in the security key, and *authenticate*, which lets the user authenticate into the website using the security key previously registered.

[5]https://www.yubico.com/products/security-key/
[6]https://fidoalliance.org/

Fig. 1: The Security Key NFC, with the USB Interface and the Capacitive Touch Sensor on top of the key.

A. Registration Protocol

At the beginning of the registration, the user asks the website to add their new security key. The website (See Fig. 2) sends to the Client a challenge, which is a long enough random number generated to avoid man-in-the-middle attacks (the Client in this case is the Browser used by the user to authenticate).

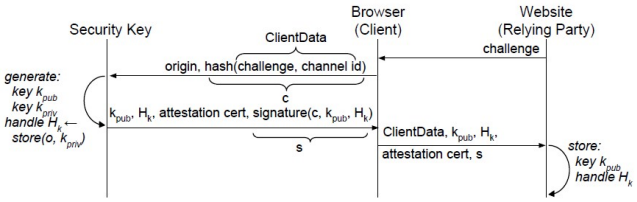

Fig. 2: Security Key registration

The Client, in turn, extracts the web origin (the website url) and hashes the ClientData. For the purpose of my sample web application, I will not cover the ClientData, but you can find more information at Lang et al. [4]. I also used their illustrations for the message exchange of the registration and authentication protocols.

The web origin and the ClientData are then sent to the security key by the Client, and the security key generates a pair of asymmetric keys, plus the key Handle H_k. I will cover later the key Handle. As for now, assume that the security key stores the web origin and the private key for future use.

The security key then returns to the Client the public key, the key Handle and an attestation certificate, with in addition a signature over: 1. the hash sent previously by the Client, 2. the public key, 3. the key Handle, and 4. the web origin.

The Client simply forwards those information to the website, which will be able to verify the signature with the public key and verify wth the attestation certificate if the key is actually trustworthy.

B. Authentication Protocol

The authentication protocol is very similar to the registration. See Fig. 3 for details.

When the user requests for authenticating, they will usually be prompted with a username-password interface, and afterwards a security key as second factor will be asked.

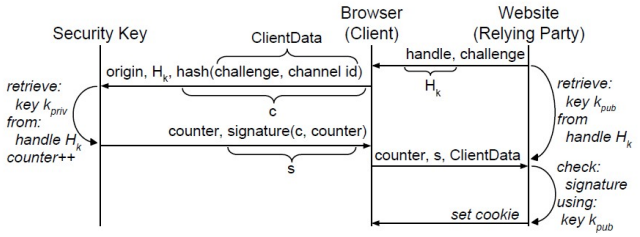

Fig. 3: Security Key authentication

At this point, the website will send the challenge and the key Handle previously stored in the database. The Client will forward the same information of the registration along with the key Handle to the security key

Now the security key will use the private key for generating the signature, and the web origin associated to that private key to check if the website requesting the authentication is trustworthy, thus avoiding real-time phishing attacks, as I mentioned in Sec. I [3]. The security key will return to the Client the signature and a counter, which is always incrementing and is used for avoiding cloning attacks. I will explain all these advantages more in detail in Sec. III-D.

The Client will forwards the aforementioned information to the website, which will in turn verify if the security key is valid. In case the website finds out the key has been cloned, it invalidates it forever.

C. Store and Retrieve Operations

Those security keys are small embedded systems which only provides cryptographic measures for generating keys and computing encryption and decryption. This is why they are designed to avoid storing information. When I wrote in Sec. III-A to assume that the security key stores the private key and the web origin associated with it, it was a simplification. If the key stored all the websites used for authenticating, there would be to problems:

1) The user would lose privacy, as their security key could be used to track their habits. In fact, habits information can be extracted by looking at which are the websites usually used by the user.
2) The user should constantly check if the security key has run out of memory. Moreover, considering the embedded nature of the device, those technologies usually do not have much memory inside it. This would be a huge usability issue.

To avoid those problems, the device leaves the website the job of storing information, harnessing a key Handle generated in registration phase. The key has two private keys that are never disclosed and that are not used for asymmetric cryptography. Those keys are used for generating the key Handle, which therefore the website cannot extract information from.

The STORE (See Alg. 1) function takes as inputs the private key k_{priv} just generated during the registration phase, and the web origin o. It encrypts the web origin using one of the two keys never disclosed (which we will call k_o) and then interleaves k_{priv} with the origin encrypted. The resulting string is encrypted again with the second key (called k_{wrap}), generating the key Handle H_k.

Algorithm 1 Store function

function STORE(k_{priv}, o)
  $o' \leftarrow Encrypt(o)_{k_o}$
  $plaintext \leftarrow Interleave(k_{priv}, o')$
  $H_k \leftarrow Encrypt(plaintext)_{k_{wrap}}$
  return H_k
end function

The RETRIEVE function will take as inputs the web origin o and the key Handle H_k previously stored by the website. Similarly to the STORE function, it uses k_o and K_{wrap} to encrypt and decrypt where necessary. More details are shown at Alg. 2

Algorithm 2 Retrieve function

function RETRIEVE(H_k, o)
  $o' \leftarrow Encrypt(o)_{k_o}$
  $(K_{priv}, o'') \leftarrow Denterleave(plaintext)$
  CONSTANT-TIME CHECK(O' == O")
  return k_{priv}
end function

I would like to have the reader focused on the advantages behind generating the H_k. Being encrypted with never disclosed keys, the website cannot retrieve any information from the key Handle, hence the user privacy is ensured. In addition, the key Handle associates each private key to the correspondent web origin. Therefore, during the authentication, the device is able to compare the website's web origin with the web origin associated to the private key. If the URKs appear different, the device recognizes a potential real-time phishing attack and drops the authentication. In this way the key is able to "store" potentially an infinite amount of websites.

D. Security Key Advantages

There are several advantages behind using a Security Key over other OTPs technologies, such as smartphones or messages, and I outlined them in Sec. II. Now, after outlining the protocol, I would like to explain how the messages exchanged denote those advantages [4]:

- **CHALLENGE:** This information is a random number used in many protocols to avoid man-in-the-middle attacks. It is actually a standard method for any protocol in the web, but it is worth to mention.
- **Key Handle:** As I explained in Sec. III-C, the Key Handle gives the user more privacy because the websites cannot be tracked on the device, and in addition solves the problem of having a limited amount of memory.
- **Attestation Certificate:** To let the websites know that the security key is a device trustworthy and that it reflects the FIDO2 and U2F protocol, there must be some check on

the validity of the hardware. However, a unique identifier would be too privacy intrusive for the user. To solve this issue, the attestation certificate of the security key is released with batches. In this way, a subset of devices is identified with each attestation certificate and the user cannot be tracked, unless they are the only ones who bought that precise batch.

- **COUNTER:** The device has an internal counter that gets incremented each time the user authenticates into a website. The counter is supposed to be always incremental, so that if the key gets cloned, the website can detect a decrement of it and, if so, invalidate that security key.
- **Test of User Presence (TUP):** Each device has a sensor that can detect if the user is present in the moment they are authenticating. The device I used 1 has a capacitive sensor on the top, and other keys have even a fingerprint scanner[7]. In those cases, the "to have" and "to be" factors are combined together.

IV. Sample Web Application Implementation

I developed a web application harnessing the u2f-ref-code library "u2f-api.js"[8] library developed by Google [4]. I took advantage of the tutorial of The Polyglot Developer[9] on YouTube, and I used the Express framework for the backend, mentioned in Sec. I. I used VueJS for the frontend, harnessing the Caddy framework. I used an old version of Caddy, it does not work if you download the latest. I detailed how to download the necessary and how to setup the sample application on GitHub[10].

Just as a reminder, this is a sample application. It is not thought to scale for bigger purposes, but it is thought to show how to harness existing libraries to relatively easily develop a security key authentication. Therefore, it does not have a database but it simply stores the user information on a variable that gets overwritten at each registration.

The U2F protocol requires to work on `https` web pages, hence I also generated a self-generated certificate to work in localhost.

A. Server Side

On the server side, I support four requests: a GET and a POST request for the registration and a GET and a POST request again for the authentication.

I used a library provided by the NodeJS environment called U2F to harness its functions `request`, `checkRegistration` and `checkSignature`. On GitHub there are the details on how to download each library I used.

1) Server Registration: See Fig. 4 for more details. When the user asks for registering a new security key, they perform a GET request to the server. The server will, in turn, use the `request` function to provide to the client an object with 1. the challenge, 2. the web origin (which is called appId in my application) and 3. the version of the U2F protocol (which in my case is the version 2).

```
app.get('/register', (request, response, next) => {
    request.session.u2f = U2F.request(APP_ID);
    /* THE REQUEST FUNCTION RETURNS AN OBJECT LIKE THIS:
    request.session.u2f = {
        version: "U2F_V2",
        appId: appId,
        challenge: toWebsafeBase64(crypto.randomBytes(32))
    };
    */
    response.send(request.session.u2f);
});


app.post('/register', (request, response, next) => {
    var registration = U2F.checkRegistration(request.session.u2f, request.body.registerResponse);

    if (!registration.successful) {
        return response.status(500).send({ message: error });
    }
    user = registration;

    console.log(user);
    response.send({ message: "Hardware key registered! "});
});
```

Fig. 4: Server registration code

The Client will forward the information provided by the server to the device, which will generate a key pair and get back to the Client with the information listed in Sec. III-A. The security key produces a Registration Message, explained in Sec. IV-B1, which will be forwarded by the Client to the server. The server will compare the Registration Message (`registerResponse` in my application) to the object previously sent to the Client to check the presence of man-in-the-middle attacks, as explained in Sec. III-A, and, if nothing is wrong, will register the new user in its database.

2) Server Authentication: Refer to Fig. 5 for details. The process is very similar to the registration. Now, the server will send the key Handle previously stored along with the other information sent mentioned in Sec. IV-A1.

```
app.get('/login', (request, response, next) => {
    request.session.u2f = U2F.request(APP_ID, user.keyHandle);
    response.send(request.session.u2f);
});

app.post('/login', (request, response, next) => {
    var success = U2F.checkSignature(request.session.u2f, request.body.loginResponse, user.publicKey);
    response.send(success);
});
```

Fig. 5: Server authentication code

In turn, the Cient will ask the device to check everything and extract the privat key, as explained in Sec. III-B. The device will return an Authentication Message, detailed in Sec. IV-B2, to the Client, which will forward it to the server.

Finally, the server will check the signature using the public key of the user previously stored and will compare the object previously sent with the Authentication Message (Called `loginResponse` in my application).

B. Client Side

The Client side of the code harnesses the Google aforementioned API, which I manually imported in the sample application.

The API provides the method `register` and `sign`, which are used by the Client to tell the device what to do [4].

[7] https://www.yubico.com/products/yubikey-bio-series/
[8] u2f-api.js
[9] https://www.youtube.com/watch?v=9d-mp6_vVUM
[10] https://github.com/davide-giacomini/u2f-login_yubikey_express

1) Client Registration: Refer to the code at Fig. 6 below. During the registration phase, the Client will get from the server the information listed in Sec. IV-A1 and will forward this information to the device. The `register` function takes as inputs the web origin (which is the object `result.data.appId` in my application), and the entire object passed along by the server (`result.data`).

```
export default {
  name: 'U2FComponent',
  methods: {
    register() {
      if (window.u2f && window.u2f.register) { // True if the browser is supported
        axios({ method: "GET", url: "https://localhost/register", withCredentials: true}).then(result => {

          window.u2f.register(result.data.APP_ID, [result.data], [], response => {

            axios({ method: "POST", url: "https://localhost/register", data: { registerResponse: response },
                    headers: { "content-type": "application/json" }, withCredentials: true }).then(result => {
                      console.log(result.data);

            }, error => {
              console.error(error);
            });
          });
        }, error => {
          console.log(error);
        });
      }
    },
```

Fig. 6: Client registration code

The device will the return the Registration Message mentioned earlier. This message contains the user public key, the key Handle, the attestation certificate and the signature over some information (Fig. 7). The Registration Message will then be forwarded by the Client to the server, which will use it under the name `registerResponse`, as seen in Sec. IV-A1.

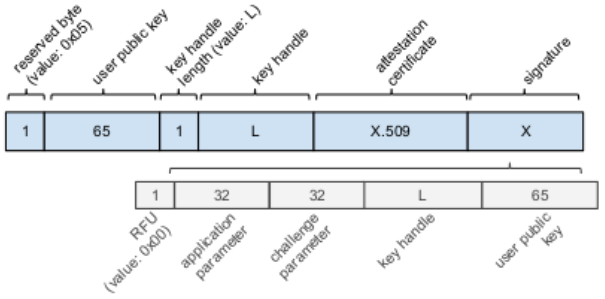

Fig. 7: Registration Message

2) Client Authentication: Refer to Fig. 8. The pattern is identical to the registration phase. Here, the function `sign` takes as inputs the web origin (`result.data.appId`), the challenge (`result.data.challenge`), and the entire object sent by the server (`result.data`).

The security key then returns the Authentication Message mentioned earlier (Fig. 9), which contains the counter and the signature over some parameters, among them the Test of User Presence.

V. Future Work

As Google says under the Chrome Extension for U2F protocol, *"support for this compiled U2F Chrome extension is being formally deprecated. No further updates or enhancements are planned, and migration to using the support built into Chrome or migration to WebAuthn is recommended"*[11]. This is also the motivation under manually importing the file `u2f-api.js` into the client-side of my application.

[11] https://github.com/google/u2f-ref-code/tree/master/u2f-chrome-extension

```
    login() {
      if (window.u2f && window.u2f.sign) { // True if the browser is supported
        axios({ method: "GET", url: "https://localhost/login", withCredentials: true}).then(result => {

          console.log(result.data);
          window.u2f.sign(result.data.appId, result.data.challenge, [result.data], response => {

            console.log(response);
            axios({ method: "POST", url: "https://localhost/login", data: { loginResponse: response },
                    headers: { "content-type": "application/json" }, withCredentials: true }).then(result => {
                      console.log(result.data);

            }, error => {
              console.error(error);
            });
          });
        }, error => {
          console.log(error);
        });
      }
    }
```

Fig. 8: Client authentication code

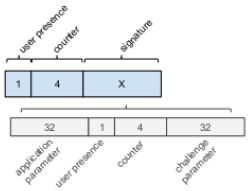

Fig. 9: Authentication Message

Future works with authentication using security keys should address the standard WebAuthn [9].

In addition, this sample can be used as a template for developing meaningful web applications that harness databases and a more complex structure.

VI. Conclusion

With this report I analyzed in detail the reasons why security keys appear to be more secure than other commonly used second factor technologies.

I outlined how the YubiKey Security Key NFC works under the hood, focusing on the parts of the protocol which are more meaningful from a security and development point of view. I did not analyze the cryptographic standards, as it was not my intention.

Finally, I explained how I used those protocols to develop a small, sample web application, taking also inspiration by a brief tutorial.

My main objective was to show to the reader how convenient is to use a security key, both from the perspective of the user and the developer.

References

[1] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent Seamons. A tale of two studies: The best and worst of yubikey usability. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 872–888. IEEE, 2018.

[2] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A

framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE, 2012.

[3] Katie Kleemola John Scott-Railton. Two-factor authentication phishing from iran, 2015. URL https://citizenlab.ca/2015/08/iran_two_factor_phishing/. Accessed: December 11, 2021.

[4] Juan Lang, Alexei Czeskis, Dirk Balfanz, Marius Schilder, and Sampath Srinivas. Security keys: Practical cryptographic second factors for the modern web. In *International Conference on Financial Cryptography and Data Security*, pages 422–440. Springer, 2016.

[5] Cormac Herley, Paul C Van Oorschot, and Andrew S Patrick. Passwords: If were so smart, why are we still using them? In *International Conference on Financial Cryptography and Data Security*, pages 230–237. Springer, 2009.

[6] Robert Biddle, Sonia Chiasson, and Paul C Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Computing Surveys (CSUR)*, 44(4):1–41, 2012.

[7] Anil K Jain, Patrick Flynn, and Arun A Ross. *Handbook of biometrics*. Springer Science & Business Media, 2007.

[8] Yubico. Security key series by yubico, 2021. URL https://resources.yubico.com/53ZDUYE6/as/q4bsft-z2wi8-4m1cae/Security_Key_Series_Product_Brief.pdf. Accessed: December 11, 2021.

[9] W3C Recommendation. Web authentication: An api for accessing public key credentials level 2, 8 April 2021. URL https://www.w3.org/TR/webauthn/. Accessed: December 11, 2021.