U2F HID Implementation - Microprocessor to U2F Key

6.858 - Computer Systems Security. May 5th, 2022.

TORQUE (TAREQ) EL DANDACHI, ASHIKA VERMA, and MUHAMMAD ABDULLAH

We design and implement a U2F Key that follows the FIDO2 U2F Protocol. The key is designed to interact with WebAuthn across different browsers as a second factor authenticator device. The code is designed to be uploaded onto a teensy with one button for verifying user presence. We use two AES128 encrypted keys hard-coded upon installation, K_{wrap} and K_{app} to generate the handle and encrypt the application parameter.

1 INTRODUCTION

1.1 Overview of U2F

Universal 2nd Factor (U2F) is an open standard that strengthens and simplifies two-factor authentication using specialized USB or NFC devices based on similar security technology found in smart cards. Essentially, it adds a second layer of protection to the simple username and password that many web services employ. The user flow of trying to log in remains similar: a user logs in with their username and password. However, at any point in time, the web service can request the user for their second factor for authentication.

For our project, we designed and implemented a U2F Key which follows the FIDO2 U2F protocol. In the following sections, we explain how we implemented and interacted with each component of the protocol, including encryption, the hardware we used for the security key, communication between the hardware and client, web authentication, and future steps.

1.2 GitHub Repository

All of our code can be found in our GitHub repository, github.com/tareqdandachi/u2f. We include code which implements the U2F protocol for a Teensy 3.2 to create a security key, and code to create a website which uses the U2F protocol for credentials.

1.3 FIDO2 U2F

We focused on implementing the FIDO2 U2F protocol. For FIDO2 U2F, the user must present a security key for their second factor, which is usually a USB device. There are two flows in the FIDO2 U2F protocol, registration and authentication as show in Figure 1 and 2.

Fig. 1. Security Key Registration[1]

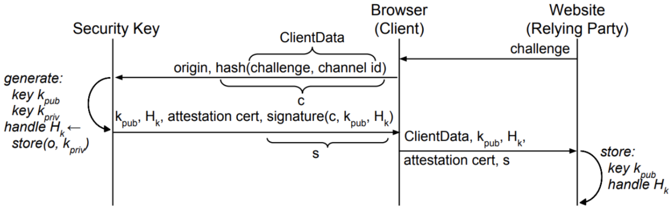
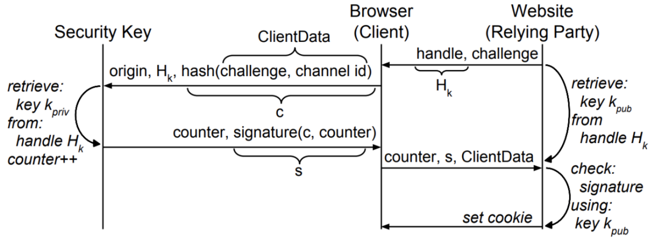

Security Key
Browser (Client)
Website (Relying Party)
challenge
ClientData
origin, hash(challenge, channel id)
c
generate: key k_{pub} key k_{priv} handle H_k ← store(o, k_{priv})
k_{pub}, H_k, attestation cert, signature(c, k_{pub}, H_k)
s
ClientData, k_{pub}, H_k, attestation cert, s
store: key k_{pub} handle H_k

Fig. 2. Security Key Authentication[1]


Security Key
Browser (Client)
Website (Relying Party)
handle, challenge
H_k
ClientData
origin, H_k, hash(challenge, channel id)
c
retrieve: key k_{priv} from: handle H_k counter++
counter, signature(c, counter)
s
retrieve: key k_{pub} from handle H_k
counter, s, ClientData
check: signature using: key k_{pub}
set cookie

There are three different components to make the U2F protocol secure, the website (or relying party), the user's browser (client), and the security key. Registration and authentication are a three step process: (1) the relying party issues a challenge to the security key, (2) the security key signs the challenge, and (3) the relying party checks the signature with the security key's public key. To prevent against Man-in-the-Middle (MitM) attacks, the origin and TLS channel ID are hashed and passed along as the application parameter to the security key, is then signed by the security key, which when passed back to the relying party, and can be verified using the public key send during registration. To prevent against device cloning attacks, the security key has a counter which increments while authenticating and the relying party can verify that the counter increased from its previous value. Lastly, there's an attestation certificate which can be re-used in multiple keys allowing clients to revoke keys if a certain model is compromised or it can be used for added checks such as a banking website only wanting their own signed security keys to be used on their website.

2 U2F PROTOCOL

2.1 Request Message Framing

The U2F Request message is framed in a standard application protocol data unit (APDU) format. This requires a 7-byte header in the following order:

- **CLA**: Always 0x00.
- **INS**: The instruction to be executed.
- **P1-2**: Two parameter bytes, **P1** and **P2**.
- **L1-2**: Length of the data to be transferred.
- **Data**: The request data.

The **INS** header byte determines whether the request is for `0x01: U2F_REGISTER`, `0x02: U2F_AUTHENTICATE`, `0x03: U2F_VERSION` or vendor specific instructions that live between `0x40` ⟶ `0xbf`. For `U2F_REGISTER`, the **P1** and **P2** fields are not used. Meanwhile

[1] Figures copied from https://css.csail.mit.edu/6.858/2022/readings/u2f-fc.pdf

P1 field is used by the FIDO Client for `U2F_AUTHENTICATE` to specify between **check-only** authentication (`0x07`) or **enforce-user-presence-and-sign** authentication (`0x03`).

2.2 Processing Registration Requests

For registration, the request data is 64 bytes long. The first 32 bytes are the challenge hash followed by the application parameter. Following the U2F Protocol, the U2F key generates a key pair and a handle from `store(application parameter, private key)`. It also generates a signature and appends it to the end.

The data is first hashed with SHA-256 and then signed with the attestation key belonging to the attestation certificate. The response is constructed as follows:

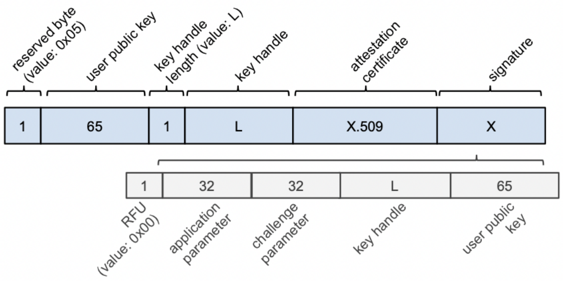


Fig. 3. Framing of a successful registration response[2]

The certificate and the signature are sent in DER format, the certificate allows the FIDO client to recognise a batch of keys so that any defective keys can be refused. The exact DER format of the signature is specified in the next section, along with the encryption used in `store` and `retrieve`.

2.3 Processing Authentication Requests

For authentication, the first 64 bytes of the request data are the challenge hash and the application parameter. This is followed by a byte specifying the key handle length and the key handle.

If the **P1** field is 0x07, we only need to verify the handle. In that case we use a modified `retrieve` operation as defined in Alg. 2. We do not return the private key, instead the return value is conditional on the assertion `app" == app'`. If the assertion passes, we respond with the error message "Test-of-user-presence required": this is the success message for **check-only** authentication, despite it being an error message in **P1**=`0x03`. If the assertion fails, we respond with the error message "Bad Key Handle".

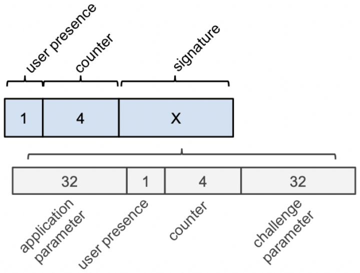


Fig. 4. Framing of a successful authentication response[2]

If the **P1** field is 0x03, we need to verify the handle and the user presence. We then follow the standard U2F authentication process; test for user presence, increment the counter and return the response in the format above. The important difference between this signature and the one used in registration is that this the signature is signed with the private key, not the attestation key.

2.4 Version Requests

The last type of request is a `0x03: U2F_VERSION` instruction request. The response message's raw representation is the ASCII representation of the string `U2F_V2`. The command takes no flags.

3 ENCRYPTION

3.1 True RNG

We seed the key generation with static to get better entropy. This is done by reading analog pin 0 and taking the least significant bit of the result. If the result doesn't change between two reads, then we count the number of cycles it takes to change and use the count's least significant bit as the next bit.

3.2 Signature

The signature is done deterministically using the relevant key. The DER format requires that the signature data (R,S pair) is appended in the following order:

- **Header**: 0x30 for compound structure.
- **Length**: The length of all that follows
- **Header**: 0x02
- **Length**: The length of the R-value.
- **R-Value**: The R-value.
- **Header**: 0x02
- **Length**: The length of the S-value.
- **S-Value**: The S-value.

We also drop the last byte of the attestation certificate.

3.3 Handle Encryption

We use the specification described in the original U2F paper for store and retrieve.

The encrypt and decrypt functions use AES-128 in 16-byte blocks on the 32 or 64 byte data, done using the Electronic Code Book (ECB) scheme. ECB is simpler than the standard Cipher-Block Chaining scheme at the cost of unreliable obfuscation on large plaintext with

[2] Figures copied from https://fidoalliance.org/specs/fido-u2f-v1.0-ps-20141009/fido-u2f-raw-message-formats-ps-20141009.pdf

Algorithm 1 Store(app, k_{priv}) → H

```
$app' \leftarrow Encrypt(app)_{K_{app}}$
$plaintext \leftarrow Interleave(k_{priv}, app')$
$H \leftarrow Encrypt(plaintext)_{K_{wrap}}$
return $H$
```

Algorithm 2 Retrieve(app, H) → k_{priv}

```
$app' \leftarrow Encrypt(app)_{K_{app}}$
$plaintext \leftarrow Decrypt(H)_{K_{wrap}}$
$(k_{priv}, app'') \leftarrow Deinterleave(plaintext)$
assert $app' == app''$
return $k_{priv}$
```

patterns. That is not an issue in our case since we encrypt at most 4 blocks of data alongside interleaving which further reduces detectable patterns.

4 HARDWARE

4.1 Microcontroller

We developed and tested this implementation on a Teensy 3.2. We based this on 3 main factors:

1. The Teensy 3.2 has a built-in EEPROM memory we can use to store the counter that retains memory even when shut off.
2. It is capable of full-speed HID communication using RawHID which is used to send 64 bytes per packet.
3. It is small and has accessible solder pads to add our own connections to the USB power and data bus.

To use the microcontroller as an HID device that interacts with WebAuthn, we need it to be discoverable as a FIDO device. This identification is done by defining a control usage defined by changing the usage page and usage id parameters of our device. The FIDO standard requires our usage page to be defined as `0xF1D0` and usage id as `0x01`. These can be changed by modifying the `usb_desc.h` file that corresponds to the RawHID protocol in the `teensy3` source.

4.2 Physical Key with Button

For our physical device, we designed a small key with a male USB A port and a button for verifying user presence. The button is in a pull down configuration with the electronics placed on a protoboard that is soldered onto the teensy body. The button is connected to pin 19 on the microcontroller. We soldered the male USB A port onto the solder pads below the teensy 3.2 corresponding to $D+$, $D-$, power and ground. The device we ended up making is displayed in figure 5.

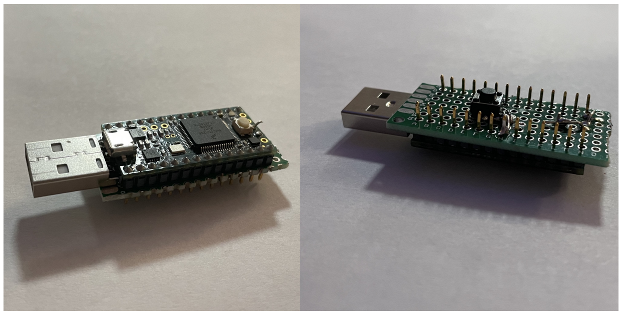


Fig. 5. Teensy 3.2 with both a male USB A port and a female micro-USB B port. The teensy has a protoboard attached to one side that houses a button and a 1K resistor in a pull down configuration.

5 COMMUNICATION

5.1 RawHID and U2FHID

For robust bidirectional communication with USB host applications, we use RawHID. We implement the FIDO2 U2F protocol over USB HID (Human Interface Device)[3]. HID transfers use smaller data chunks per transaction and don't require drivers causing them to be more reliable than USB CDC (Communications Device Class) transfers (i.e. virtual serial ports).

Our hardware supports sending and recieving 64-byte packets with a host application, in this case a WebAuthn client. Since it doesn't require drivers, it works out of the box and should automatically be detected by browsers that implement the WebAuthn API. This transfer method allows for multi-application support, fixed latency and low-overhead; making the U2FHID protocol much more scalable for discovery and use. When the HID communication begins, our hardware sends parameters over that specify the vendor id, product id, usage page and usage id (discussed in section 4.1).

Messaging in this protocol consists of three layers. Firstly, we need a way to carry HID reports at the channel level and encode messages inside of these transactions. This level of messaging contains instructions that start, end and communicate within the HID scheme. They are also responsible for channel allocation and propagating errors at the HID level. The second layer focuses on Application Protocol Data Unit (APDU), a way to encode response and request data along with status words. The final layer deals with the representation of response and request data in a way pertaining to our protocol, this is discussed in section 2.

5.2 Communication Protocol/Transactions

5.3 Packets and Channels

The U2FHID protocol defines transactions, which are requests followed by a message. Transactions are blocking once intiated: once a transaction is initiated, it needs to be fully resolved before a new transaction takes place. Messaging is done through a smaller unit of data called a packet, these correspond to HID reports. Packets have a fixed size.

The protocol is designed for concurrency: multiple clients can access a single resource through the HID stack. The protocol defines logical channels (channels handled through encoding rather than

[3] https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.pdf

physically different channels). To manage routing, our implementation assigns one of four unique channel identifiers to each application. Channel identifiers are 32-bit numbers where `0x00000000` and `0xffffffff` are reserved. `0xffffffff` is used for channel allocation.

To manage concurrency, we keep track of 5 internal states: `Available`, `Wait_init`, `Wait_cont`, `Timeout`, `Large`. This helps our device manage concurrency with requests, enter busy modes and allow for consistency between the processing requests.

5.4 Messaging and Headers

Each request and response begins with an initialization packet and is followed by continuation packets if the messaging requires more space. The U2F device never sends a response if no request is initiated from a host.

5.4.1 Initialization Packets. The first 32 bits of all packets are a channel identifier. Next follows 8 bits corresponding to a command identifier. The command identifier encodes different U2FHID commands such as `U2FHID_MSG`, `U2FHID_INIT` and `U2FHID_ERROR`. The next 16 bits encode the payload length. The remaining 57 bytes are the payload. The command identifier always ahs the highest bit set to 1 (command identifier always greater than `0x7f`), to distinguish it from a continuation packet which has that bit always set to 0.

5.4.2 Continuation Packets. As with initialization packets, the first 32 bits encode the channel identifier. This is followed by a bit that is always set to zero, identifying the packet as a continuation packet. The next 7 bits encode the packet sequence which starts at 0 and is incremented for every next packet. This is then followed by 59 bits corresponding to the payload. Note that this means with our implementation of 64-bytes of HID communication (the maximum for full-speed devices), the maximum payload length is 7609 bytes.

5.5 Message Framing

5.5.1 Request Message APDU. The request message is encoded using the Application Protocol Data Unit (APDU) spec and needs to be processed to extract the different parts of it. The APDU can be encoded in Extended Length Encoding for a maximum length of 65,535 bytes or a Short Encoding allowing a maximum of 256 bytes to be sent. The message payload is prefaced with a CLA byte which we set and check to be zero. It is then followed by an instruction code which contains U2F protocol instruction codes that specify what part of the U2F protocol to run. This is followed by 2 parameters defined by the U2F instruction used. It is followed by the length of the request data and expected length of the payload.

5.5.2 Response Message APDU. The response message ADPU contains the response data followed by a 16-bit status word. The U2F protocol defines 6 status words as defined in ISO7816-4 with a special meaning:

(1) `0x9000 SW_NO_ERROR`: command completed successfully.
(2) `0x6985 SW_CONDITIONS_NOT_SATISFIED`: failed test of user presence.
(3) `0x6A80 SW_WRONG_DATA`: invalid key handle.
(4) `0x9000 SW_WRONG_LENGTH`: invalid length of request.
(5) `0x6E00 SW_CLA_NOT_SUPPORTED`: "Class byte of the request is not supported." In our implementation, CLA byte is enforced to be `0x00`.
(6) `0x6D00 SW_WRONG_LENGTH`: requested instruction not supported, our implementation only supports the main 3 outlined in the U2F spec with no other vendor specific instructions: `U2F_VERSION`, `U2F_REGISTER` and `U2F_AUTHENTICATE`.

6 WEBAUTHN

6.1 WebAuthn API

The Web Authentication API (WebAuthn) is a specification written by W3C and FIDO with participation of Google, Mozilla, Microsoft, Yubico and others. It is a public key extension of the existing credential management API. Essentially, it allows servers to register and authenticate users using the FIDO2 U2F framework and makes it easy for servers to add this additional layer of security to the servers current password management system. In addition to the supporting FIDO U2F devices, WebAuthn also allows other types of authenticators such as Android Attestation Keys or Trusted Platform keys (like the fingerprint sensor on some MacBooks).

6.2 Registration

The client first issues a request to the relying party to register along with user information for the relying party to store credential information about the user. Next, using WebAuthn, the relying party can issue a 32 byte challenge and pass along parameters such as relying party url or the type of attestation. From there the browser issues requests for a second factor to register. The user can press their FIDO-U2F compliant security key to run through the protocol detailed in Section 2. The Webauthn API creates a PublicKeyCredential object which contains the `ClientData` (buffer) and attestation object (buffer). The client then send the `PublicKeyCredential` to the relying party or server. The `PublicKeyCredential` can be decoded and unravelled to get the `ClientData`, public key of the security key, the handle, the attestation certificate and signature, which the relying party uses as shown in Figure 1. The relying party can check the signature with the public key and if needed verify the attestation certificate. The relying party then stores the public key of the security key and handle in its own database. From there, the relying party can tie a user to these new credentials by saving these credentials.

6.3 Authentication

Authentication follows a similar flow as registration using Webauthn, except the security key sends back a counter and a signature of the counter plus the hash of the `ClientData`. The relying party then checks the signature with the private key that it stored earlier and also checks if the counter has increased from the previous observed value to prevent device replication attacks. After authentication, the relying party can log the user in for the rest of the session.

6.4 WebAuthn Requests

The WebAuthn API communicates with keys using requests as discussed in section 5. Once a host triggers a WebAuthn authenticate message, the message is broadcast to all connected U2F devices.

The first device to respond with a positive test of user presence allows the authenticator to continue with the communication. The broadcast to all connected U2F devices is filtered by the usage page that corresponds to that of the FIDO2 U2F protocol as discussed in 4.1.

6.5 Testing HID to WebAuthn Communication

We created our own relying party which uses WebAuthn which can be accessed at u2f-858.herokuapp.com. However, other useful resources to verify the entire stack is functional included:

- webauthn.io
- demo.yubico.com/webauthn-technical

7 RESULTS AND FUTURE STEPS

Our implementation worked on registering and authenticating on modern websites and is secure against testing on the Yubico Key testing suite. We published our implementation onto GitHub. The implementation works as-is with little to no hardware modification (the addition of a button for user presence but that can be turned off).

Given more time, we would like to develop changes on the hardware. Namely, we would want to add a true random number generator hardware module that we sample from instead of seeding randomness by measuring static. We would also design hardware specifically for encryption and decryption on a secure element. We hope to be able to convert this implementation into robust hardware rather than software running on a microprocessor to remove the reliance on a secure infrastructure provided by the teensy.

On the software side of things, we hope to emphasize the accessiblity of this implementation so that anyone could load this onto any microcontroller they have. We designed and tested this on a Teensy 3.2, but there is nothing that is completely specific and not reproducible on other hardware (except for counter on EEPROM on some devices). Hopefully this could turn into a user friendly program people can use to make a custom key without needing to install arduino.

REFERENCES

(1) https://fidoalliance.org/specs/fido-u2f-v1.0-ps-20141009/fido-u2f-hid-protocol-ps-20141009.html
(2) https://fidoalliance.org/specs/fido-u2f-v1.0-ps-20141009/fido-u2f-raw-message-formats-ps-20141009.pdf
(3) https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-hid-protocol-v1.2-ps-20170411.pdf
(4) https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-raw-message-formats-v1.2-ps-20170411.html
(5) https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-client-to-authenticator-protocol-v2.0-id-20180227.html
(6) https://fidoalliance.org/specifications/
(7) http://www.linux-usb.org/usb.ids
(8) https://github.com/tonijukica/webauthn.git
(9) http://bitcoin.stackexchange.com/questions/12554/why-the-signature-is-always-65-13232-bytes-long
(10) https://css.csail.mit.edu/6.858/2022/readings/u2f-fc.pdf
(11) https://demo.yubico.com/webauthn-technical/
(12) https://webauthn.io/