



# Evaluation and Implementation of SQRL and U2F as 2<sup>nd</sup> Factor Authenticators for CERN Single Sign-On

**September 2015**

Author:  
Azqa Nadeem

Supervisors:  
Vincent Brillault  
Stefan Lueders

CERN Openlab Summer Student Report 2015



## Project Specification

The project focuses on evaluation and implementation of SQRL (Secure Quick Reliable Login) [29] and U2F (Universal 2<sup>nd</sup> Factor) [29] protocols for 2<sup>nd</sup> Factor Authentication modules at CERN. This report presents a summary of the critical analysis of the two protocols and presents implementation details of U2F as a 2<sup>nd</sup> Factor.

# Abstract

Secure Quick Reliable Login (SQRL) and Universal 2<sup>nd</sup> Factor (U2F) are two new strong authentication protocols. They both operate on a challenge-response model and use Asymmetric key encryption.

SQRL aims to replace user names and passwords because they are our identity and we cannot trust the websites to keep our personal information safe. In order to authenticate against a SQRL aware service, one has to use the SQRL application on his phone. A perk of using SQRL is that the users only need to remember a master password for the SQRL application itself rather than passwords to all the different authentication services they use.

U2F, on the other hand, is hosted by the FIDO alliance. It has been adapted by big companies, such as Google, Visa, Yubico, etc. It requires a physical U2F enabled token on the client side and a U2F aware authentication service on the server side.

After an evaluation of the two protocols, we have come to a conclusion that at this point, U2F is a better option than SQRL. Hence, a web application has been implemented and deployed on CERN web servers to demonstrate the functionality of U2F.

## Table of Contents

1	Introduction .....	5
1.1	Multi Factor Authentication (MFA) .....	5
1.2	One Time Password .....	6
1.3	Challenge-response mechanism .....	6
2	Secure Quick Reliable Login (SQRL) .....	8
2.1	User Interaction .....	8
2.2	Technology and Security .....	8
2.3	Implementation Requirements .....	11
2.4	Comparison and Observation .....	11
2.5	Conclusion .....	12
3	Universal 2 <sup>nd</sup> Factor (U2F) .....	13
3.1	User Interaction .....	13
3.2	Technology and Security .....	13
3.3	Implementation Requirements .....	15
3.4	Comparison and Observation .....	15
3.5	Conclusion .....	16
4	U2F Implementation .....	17
4.1	Server .....	17
4.1.1	Register API .....	18
4.1.2	Authenticate API .....	19
4.2	Clients .....	20
4.2.1	Web Browser .....	20
4.2.2	C# Handler (Host Library) .....	21
4.3	Persistent Storage .....	22
4.4	Deploying .....	23
4.5	The Bigger Picture .....	24
5	Virtual Walk-through of User Interaction .....	26
6	Challenges Faced .....	27
7	Conclusion .....	28
8	Glossary .....	29
9	References .....	30

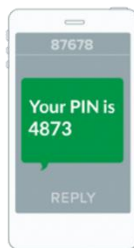
## Table of Figures

Figure 2-1:	SQRL key generation .....	9
Figure 2-2:	SQRL token registration .....	9
Figure 2-3:	SQRL token authentication .....	10
Figure 2-4:	MITM scenario .....	11
Figure 3-1:	U2F token registration .....	14
Figure 3-2:	U2F token authentication .....	14
Figure 4-1:	Application architecture .....	17
Figure 4-2:	Register API .....	19
Figure 4-3:	Authenticate API .....	20
Figure 4-4:	Web - Registering a token .....	20
Figure 4-5:	Web - Authenticating a token .....	21
Figure 4-6:	Handler - Registering a token .....	22
Figure 4-7:	Handler - Authenticating a token .....	22
Figure 4-8:	Database Schema of the project .....	23
Figure 4-9:	Overall architecture .....	25

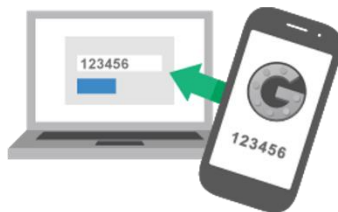
# 1 Introduction

The concept of Multi-Factor Authentication [29] requires the client to know a secret (username/password) and to possess a secret (physical token, code on cell phone, etc.) in order to get authenticated to a service. Ever since this concept was introduced, there have been many different protocols introducing different features to authenticate users more securely. Many services now provide more than one 2nd Factor Authentication mechanism to allow their users to be able to authenticate themselves with more flexibility.

CERN currently offers four modes of 2<sup>nd</sup> Factor Authentication [29], namely: **SMS based**, **Google Authenticator** (software based), **YubiKey** and **Smartcards** (hardware based). This report introduces two new protocols for 2<sup>nd</sup>FA, namely: **Secure Quick Reliable Login (SQRL)** and **Universal 2<sup>nd</sup> Factor (U2F)**. This report evaluates them with a special emphasis on security and cost effectiveness and then provides implementation details for the protocol that seems to be the most feasible one.



SMS



Google Authenticator



Yubikey



Smartcard

## 1.1 Multi Factor Authentication (MFA)

Multi Factor Authentication (MFA) is a mechanism for access control in which the user has to go through multiple stages to fully get authenticated to a service. The stages can require you to either know a secret (knowledge factor) or to be in possession of one (possession factor). These several independent credentials allow a user to access the locked resources.

MFA adds another layer of security to protect the users from online identity theft. It has been known for years that password based authentication mechanisms are not the most secure way to authenticate oneself. Moreover, while registering to an authentication service, we give the server a shared secret to keep, which it does not always store in the most secure fashion. In that case, an attacker might be able to grab this shared secret at any time. Hence, it is important to introduce a method by which just the knowledge of the shared secret would not be enough to authenticate oneself to a



service. The other information that completes the process must come from the real owner of the account, so it can either be a physical token or a random code on a device that only the real owner possesses.

## 1.2 One Time Password

One Time Passwords (OTPs) [29] are pseudorandom values that are valid for one time use only. It can either be a Time variant OTP (TOTP) or HMAC based OTP (HOTP). In the case of HOTP, every time an OTP is used, the client and server increment a counter, which keeps them in sync. The counter also contributes in the generation of OTP so it is unique and the server can easily verify which value to expect from the client. TOTP is similar to HOTP but the counter is replaced with a time stamp, which is used to generate the OTPs. It is important that the clocks of the server and the client are synced, otherwise the server will not be able to verify the OTP sent by the client. The algorithm and the seed used for OTP generation is agreed upon beforehand.

Many attacks can be thwarted if we use OTPs in addition to usernames and passwords. Because of the randomness and non-reusability of the OTPs, dictionary and replay attacks would almost always fail. However, there are some downsides of using OTP. If you are using SMS based OTP and you are in an area where you don't have coverage, you will be locked out of your account because the OTP will never reach you. Also, one OTP cannot be used with multiple accounts because the seed for the generation of OTP will be different for different servers. The server and the client can become de-synchronized, such as a bad clock in case of TOTP and generation of too many hashes for HOTP that never reached the server, in which case the authentication mechanism can suffer badly.

## 1.3 Challenge-response mechanism

In a challenge-response mechanism, the authenticating party sends a challenge (nonce) to the requesting party. The requesting party signs the challenge and the authenticating server has to verify the signature to authenticate the identity of the requester.

There can be two scenarios in challenge-response mechanism – either the server authenticates the client (Unilateral authentication [29]) or both the server and client mutually authenticate each other (Mutual authentication [29]).

For both cases, there is no shared secret – only the corresponding public key needs to be distributed at the time of registration. The client sends a request to the server. The server replies back with a time-variant random number, which we call as a challenge or nonce. The client signs the challenge with its private key and sends back to the server. The server verifies the signature with the public key of the client and finally, if the identity is authenticated, the client is given access to the locked resources.

In the case of Mutual Authentication, there is an added step of verifying the server as well. So, the client sends another challenge to the server. The server signs the challenge with its private key and sends back to the client. The client verifies the signature with the public key of the server and if the server is authenticated, the session is initiated.

When considering about the security and authenticity of the public key, and to protect oneself from man-in-the-middle attack, where the public/private key pairs can be swapped, an important aspect to take into consideration is ensuring a correlation between the public key and the actual identity of the owner of the private key. Hence, the server maintains an Identity Management System, either using LDAP or some other mechanism to keep track of identities of its clients.

## 2 Secure Quick Reliable Login (SQRL)

SQRL [30], the brainchild of Steve Gibson, is a software based authentication system that aims to replace usernames and passwords altogether. The idea behind SQRL is that we give our personal information (usernames, among other information) to servers to keep it safe, while the servers don't store that information in the most secure fashion. Instead, the developers propose that instead of giving any "significant" information that can potentially expose your identity over the Internet, the server only has to store a set of public keys, which, even if lost, will pose minimal threat to the identity theft of the user. Hence, they propose a new mechanism through which, you only have to scan the QR code sent by the server and you'll be logged in. Your entire identity over the Internet is derived from a master key that is stored in your SQRL application and that key is protected by a master password. This way, the users only need to remember that one master password for the SQRL application itself rather than multiple passwords for all the different services they use.

### 2.1 User Interaction

The SQRL application is installed on the user's device while the server that the user tries to log on to is SQRL aware as well. It is crucial for any production deployment that the SQRL client application is available for all major platforms e.g. iOS, Android, Windows, Linux etc.

Whenever a user tries to authenticate him/herself to the website, the authentication service will send a challenge (a securely generated long random nonce) encoded into a QR code, which the user can either scan (using QR code reader), tap (on touch screens) or click (on phones or desktop). The SQRL application takes care of the rest of the steps and finally, the user is logged in.

For web authentication, the user can either scan, or click on the QR code. In case of non-web authentication, when the QR code cannot be displayed to the user, the URL can be directly shown and the user can click on it, or use the keyboard to activate the link.

### 2.2 Technology and Security

SQRL uses asymmetric key cryptography to provide security. The client has one master key that represents his/her entire identity over the Internet. A public/private key pair is generated using HMAC (SHA-256). The master key and the domain name (the website to which the user is trying to authenticate into) are hashed to derive a private key, which is further manipulated to derive a matching public key. Each site has a different public/private key pair for different users but the same user will always have the same public/private key pair for the same site because the master key and domain name never change.





Figure 2-1: SQRL Key generation

For registration, the user sends a request to the server. The server replies back with a long securely generated challenge encoded in a QR code. The SQRL app scans the code, generates public/private key pair for that site, signs the challenge with the site specific private key and then sends a POST request to the server containing the site specific public key and signed challenge. The server verifies the signature using the public key and then stores the public key for authenticating the user in the future.



Figure 2-2: SQRL token registration

After the user is registered to a website and the generation of the site-specific key pair, whenever the user tries to authenticate him/herself to the website, the authentication service will send a challenge (a securely long random number) encoded into a QR code, which the user can either scan (using QR code reader), tap (on touch screens) or click (on phones or desktop). The client signs the challenge with its site-specific private key (can be re-generated on the fly) and sends a POST query to the service containing the signature. The service only needs to implement a couple of methods, one of them being able to verify the signature by using the stored public key, and if it is successful in doing so, authenticate the user.

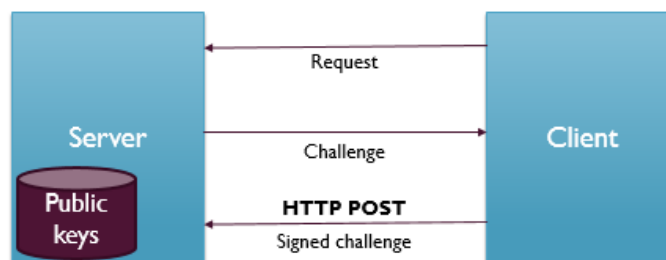


Figure 2-3: SQRL token authentication

The master key itself is the single most important thing that needs to be kept safe. For that purpose, the master key is encrypted with a password, which is the *only* password the user needs to know. SCrypt – a password based key derivation function is used to protect the master key with your SQRL client application password.

Since every nonce embedded into the QR code yields a different QR code, no replay attack is possible. Also, because of the randomness of the nonce, any brute force attack will also not be feasible for an attacker. A SQRL enabled server cannot impersonate a client because SQRL uses asymmetric key cryptography, unlike YubiKey which uses symmetric key cryptography and hence, exposes the client to be impersonated by the server. Man in the Middle (MITM) attacks are thwarted to some extent. Since the public/private key pair is site specific and if someone tries to change the domain name even slightly, the key pair would be different than the real one and the server would know that it's under attack. Also, during the authentication process, the client is shown the host name and is asked if they really want to log in to that site. If legit.com was showing a QR code embedding a nonce sent by evil.com, the client application would ask the user if they want to log in to evil.com (because the challenge came from evil.com's authentication service) and the user can easily identify the redirection but putting such a big responsibility on the user is a risk.

Consider a scenario where a user logs in to evil.com mistaking it as legit.com. evil.com then sends an authentication request to legit.com. legit.com sends back a valid QR code to the requestor – evil.com. evil.com presents it to the user. The user scans the code and authenticates him/herself to legit.com, but evil.com now has an authenticated cookie for the user's session. This mainly happens because the device where the challenge is sent and the device that responds to that challenge are not always the same (because of the hand-off functionality) and there is no mechanism on the server side to correlate the computer's IP address to the phone's IP address, nor is there any way on the client side to learn what the real remote host is, if there is proper SSL encryption, etc.



Figure 2-4: MITM scenario

## 2.3 Implementation Requirements

For the server side, we need to develop a web API that is able to generate nonces and verify signatures. It should be integrated to the service we want to log in to. For the client side, we need to develop a fully functional app that is able to create, export and import user identities and is able to scan QR codes. It should also notify the user about the host name of the website he/she is about to log in to. For SSH, the SSH server side needs to be able to receive a challenge from the server, forward it and get a response from the server.

Currently, there is no company backing SQRL. Moreover, the implementations of SQRL, which are in production, are very few and on very few platforms. There are prototypes available on their website but CERN will have to develop its own server side (one API to verify the signatures) and client side application to let its users use it. It will soon become too costly for CERN to maintain the SQRL infrastructure.

## 2.4 Comparison and Observation

The following are some of the observations and opinions about SQRL being used as a 2<sup>nd</sup>FA method at CERN:

- 1) It will need a working Internet connection on the phone for its operation unlike Google Authenticator which works in the offline mode.
- 2) It will be useful in scenarios when you are trying to log in to an untrustworthy computer and you don't want to type anything into the attached keyboard. Please note that it will not be applicable when you are using SQRL as a 2<sup>nd</sup>FA.
- 3) SQRL is relatively safe in the sense that the master key never leaves your phone. The fact that the site specific private keys are never stored on your phone is also a plus. The only key you need to protect is the master key and that you do, by encrypting it with a strong SQRL app password. The protocol uses SCrypt – a memory hard function to protect the master key so, even dictionary attacks can be thwarted to some extent.
- 4) As a 2<sup>nd</sup> factor, I believe SQRL is too much of a hassle to setup and use. If you were to use SQRL at CERN, you will still have to remember one

password, but that will be for your SQRL client app, rather than the password to your CERN account. In addition, CERN's password has a limited lifetime. In the case of SQRL, we can't impose this requirement on the public/private key pair, because it would require us to change our master key, which is our whole identity on the Internet. Hence, we will not be able to control the lifetime of the key pairs.

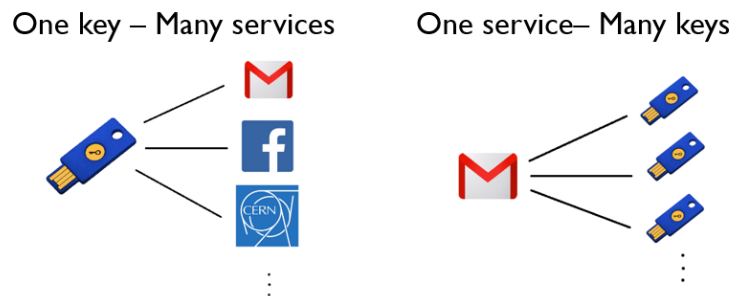
- 5) One person cannot have more than one account on the same website against the same master key. For example, I cannot sign up to Google mail client for more than one account because the public/private key pair generated from domain name and master key will always be same.
- 6) **Consideration:** It might be feasible to implement SQRL, side by side with the 1<sup>st</sup> factor authentication (username/passwords) for people who will rather not type their passwords on a public computer or when it's not secure. But, we first need to find out the future plans of the developers, e.g. whether they plan on releasing a final product on different platforms and fixing security loopholes in their protocol.

## 2.5 Conclusion

Currently, there are some serious loopholes in the protocol. It protects you from threats you are not likely to face while ignoring some of very important aspects, e.g. the Man in the Middle attack. I would conclude that we should wait for the protocol to become mature enough that it counters all the security flaws and has support for different platforms.

### 3 Universal 2<sup>nd</sup> Factor (U2F)

U2F [30] is an open protocol hosted by the FIDO Alliance. It is a physical token based, strong authentication mechanism that works on asymmetric key cryptography. It was initially developed by Google but now is hosted by FIDO Alliance. There is a many-to-many relationship between the security keys and the authentication services, which makes it convenient for the end user because then, they don't have to carry along a lot of keys – one key is able to authenticate them to many different authentication services.



#### 3.1 User Interaction

The U2F device is a physical token that will be linked to a user's account. The user just has to connect his/her device to the computer and press the button on the device to get authenticated. The interaction will pretty much be the same for the web and the CLI.

#### 3.2 Technology and Security

U2F uses strong authentication protocol and asymmetric key mechanism for authentication and security.

During the registration process on a new website, the client sends a request to the server. The server replies back with enrolment data, containing a securely generated random number, app id (domain name) and the version of the protocol. The client passes the enrolment data over to the U2F enabled token. The token generates a site specific public/private key pair, signs the challenge with the site specific private key and sends the site specific public key and signed challenge to the client, which further passes this data to the server. The server verifies the signature and stores the public key and signed challenge for authenticating the client in the future. The signed challenge is also known as a Key handle [29] which is a specially designed value. It contains the challenge, along with the app id and user information, so the device knows exactly which origins it has registered to.

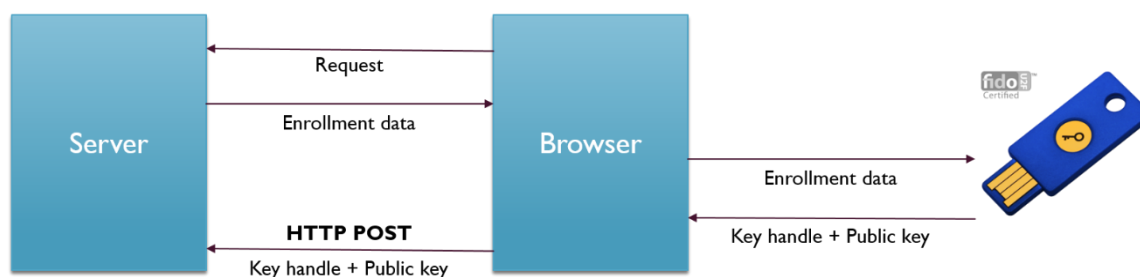


Figure 3-1: U2F token registration

During log in, the client sends a request to the server. The server sends a challenge along with the key handle to the client, which is further passed to the token. The token looks for a familiar key handle, and only if it finds one, does it sign the challenge using its site specific private key and passes it to the server. The server verifies the signature using the site specific public key.

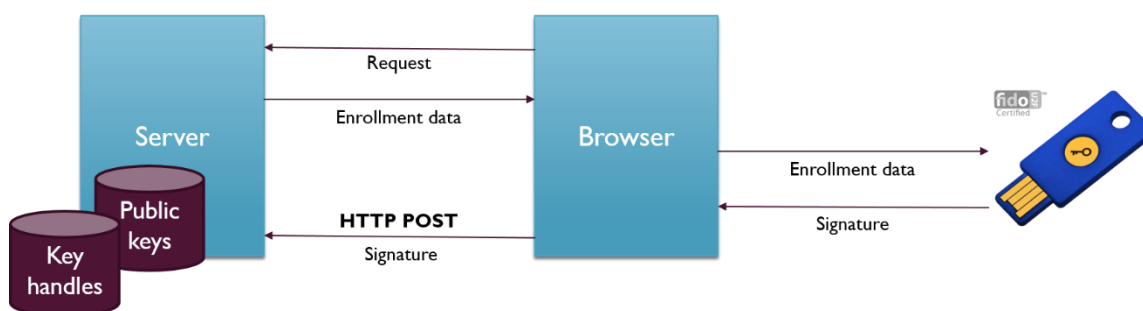


Figure 3-2: U2F token authentication

The key handle is different for different origins – websites. A U2F device will not sign just any Key handle it sees – it will only sign the one it already recognizes. So, if the Key handle doesn't match the origin name it tried to log in to, no signature will take place and hence, no phishing attacks are possible here. Also, the handle cannot be passed among different colluding sites. U2F is protected against many of the MITM attacks because of the direct involvement of the browser. The only downside is that we are trusting our computer in this case – if the computer gets compromised, our session gets stolen. This will only happen if the attacker is somewhere between our browser and the U2F device and is able to intercept that communication.

Another good thing about U2F is that it shows a dialog box asking for confirmation if the user wants to log in to a particular site (taken from the Key handle) and only if the user says yes, is the signature done and sent to the server. A *human presence* [29] is also necessary for the signature process because it will only take place once the user confirms by pressing the button on the U2F device.

U2F device also maintains a usage counter for the number of times the device is used, hence a clone of that device cannot be made. That counter is

concatenated to the client data before signing and being sent to the server. Since the server and U2F device both remember the value of usage counter, the server can easily use it to verify if the device has encountered some problem or has been cloned.

### 3.3 Implementation Requirements

For the server side, we need to develop a web API that is able to generate and keep track of the Key handles and verify signatures. This API would be split into two parts, one dedicated to register a user and another one for authenticating the user. It should be integrated to the service we want to log in to. However, for the client side, we need a U2F enabled physical device along with a client side handler— either a browser or a desktop application, that is able to act as a bridge of communication between the U2F token and the web API.

Google is supporting U2F and as a result, it has added its support in its browser, OS and mail client. Support for more platforms is also expected because many big companies are involved in it. Currently, there is full support only for Chrome. A patch for Openssh is available to integrate it with U2F. Firefox and Microsoft are also working on adding its support to their products.

### 3.4 Comparison and Observation

The following are some of the observations and opinions about U2F being used as a 2<sup>nd</sup>FA method at CERN:

- 1) U2F actually seems to be better than YubiKey in the sense that YubiKey uses Symmetric key encryption while U2F uses Asymmetric key encryption, which is far more secure. Also, for every service, you need a separate YubiKey while U2F device can store all the public/private key pairs on one device for all the different services. Users can also use one U2F device to authenticate into multiple non-CERN services and separate accounts.
- 2) There will be no need to maintain the client side once the support is available for all major platforms. The cost of maintaining the infrastructure is reduced. Its server side implementation is available on GitHub which is directly managed by Google and client side libraries managed by Google and Yubico.
- 3) We might need a device driver on the user's system, if the device is anything other than a USB e.g. an NFC or a specialized token.
- 4) U2F is better than One Time Password (OTP) sent by SMS because you don't have to wait for the code to arrive on your cell phone, when for example, your phone is out of coverage.
- 5) If an attacker tries a MITM attack, the software client and the U2F device can detect it in most cases. For example, if evil.com tries to intermediate between

the U2F device and something.com, the U2F device will never respond because of different domain name. The U2F device will also detect if a different Key handle is sent to it and the signature process will not take place.

## 3.5 Conclusion

U2F seems like a good 2<sup>nd</sup>FA method because there are less risks associated to its adoption, in comparison to SQRL. It is safe and there is server side support available that we might only need to integrate. Hence, the cost of using U2F is also reduced.



## 4 U2F Implementation

An open source implementation of the server is available on GitHub, and is maintained by Google. We have decided to use that implementation for CERN, as it will cut down the cost for maintenance.

### 4.1 Server

The open source implementation uses Google App Engine [30] as its server. They have used servlets to manage the user's session, register and authenticate tokens. Every user's account name is associated with their Google+ account. So basically, the project only implements the 2<sup>nd</sup> factor, and not the first factor because that is automatically handled by Google App Engine. This is good for CERN because CERN also handles the first factor itself and we only need a project with 2<sup>nd</sup> factor implemented.

The problem with GAE is that it only supports Google's own cloud services and provides no support for Oracle, while CERN uses Tomcat as its Java engine and Oracle as its database. Therefore, we had to adapt the code accordingly.

Google App engine is convenient in the sense that its internal support for SSL/TLS is easy to configure while Tomcat requires much more fine-tuning. Therefore, for development, we have used Nginx as a reverse proxy, to establish a secure channel. The requests are sent to the SSL enabled nginx server that redirects them to the U2F enabled application server, and the responses are forwarded to the clients through the nginx server.

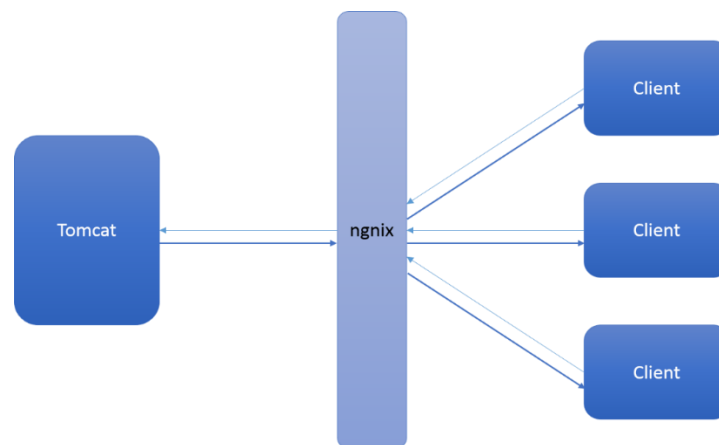


Figure 4-1: Application architecture

After deploying the web application, we now have a web API that can register U2F tokens and store their information in the database. We also have an API that

can authenticate U2F tokens and another one that can remove the registered tokens. All these APIs can be accessed over HTTPs.

### 4.1.1 Register API

As soon as the client – browser or C# handler calls the register API, a `BeginEnroll` servlet is called. This servlet takes the current username and the app id as request parameters. It generates and assembles enrolment data containing the app id from the client, a challenge and the version of the protocol used. It looks like the following blob:

```
{
  "appId": "https://test-tomcat-u2f.web.cern.ch",
  "challenge": "qzIRnWWkv87qSo7sfmYlGw",
  "version": "U2F_V2"
}
```

This data is sent back to the client, which passes it to the security key. The client waits for *user presence* by waiting for the user to press the button the token, after which the response from the key is collected. It contains *client data* and *registration data*. These values are passed as request parameters to another

```
{
  "registrationData": "BQQ0td__bgmv8V6_T-E4914xE-Pb6ji1YMUoP0LDLDCGtzCH
PwbkMLlxlo6C6fawnQ7671o85nSbek9v0m3_zSt5xV4VBgwgIbMIIBBaADAgECARxBIIMAsGCSq
GSIB3D ",
  "clientData": "eyJhY2hhbGxlbmdlIjogIjZsOGFSTTZhMzVod3JyYW1ydDdzS3Q3Z0RrdlRh
bXQyc1lyTWdNWU5cm8iLCAib3JpZ2luIjogImh0dHA6XC9cL2RlbW8ueXViaWNvLmNvbSIsICJ0eX
AiOiAibmF2aWdhZG9yLmlkLmZpbmlzaEVucm9sbG11bnQiIH0="
}
```

servlet called `FinishEnroll`. The *client data* is an encoded value containing the type of client used, and the challenge signed, etc. The *registration data* is also encoded, containing the public key, an attestation certificate, a key handle and a signature. These values are decoded at the server end, the signature is verified using the public key and then, if all works well, registration data is stored in the database. Typically, the enrolment time, the key handle, the public key, the attestation certificate and the counter are stored against the username for each key that is registered.

If the client is Google Chrome, it will communicate with the token using the built-in support. However, if it's the C# handler, a host library will be needed to communicate with the key. When the handler receives the enrolment data, it will use the following command to pass the enrolment data (stored in `enroll_data_file`) to the key, and will wait for the key's response too.

```
u2f-host.exe -aregister -o <app_id> < <enroll_data_file>
```

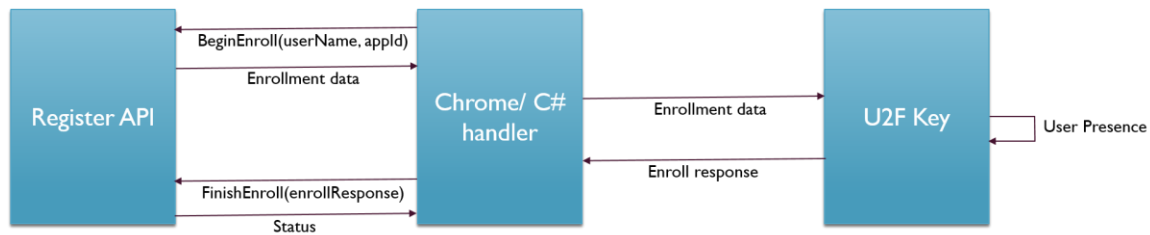


Figure 4-2: Register API

### 4.1.2 Authenticate API

When the client – browser or C# handler calls the authenticate API, the `BeginSign` servlet is called. This servlet takes the current username and the app id as request parameters. It generates and assembles the sign data containing the key handles, the app id from the client, a challenge and the version of the protocol used. It looks like the following blob:

```
{
  "appId": "https://test-tomcat-u2f.web.cern.ch",
  "challenge": "qzIRnWWkv87qSo7sfmYlGw",
  "version": "U2F_V2",
  "keyHandle": "Y8T6h5aWx9s7a8joWWSJAFVUGM4b4IIXUx6r2xFmNPcebuPL-3NPO-
PyGfgaNkWyhXppUVdZAOWhv4fGxdPRLA"
}
```

This data is sent back to the client, which passes it to the security key. The client waits for *user presence* by waiting for the user to press the button the token, after which the response from the key is collected. It contains *signature data*, *client*

```
{
  "signatureData": "AQAAAAIwRAIgPIlfE6dsRykm5M_KG88hHjRh2ZdiyMakVUIKG9Q2w9",
  "clientData": "eyJhY2hhbGxlbmdlIjogIlBhM2V1Y0ZRCkgtNWU0FFZEdFU0ppSVc5cG9f
U296czZFZlB1WU4zbk0iLCaib3JpZ2luIjogImh0dHA6XC9cL2RlbW8ueXViaWNvLmNvbSIzLm
lkLmdldEFzc2VydGlvbiIgZQ==",
  "challenge": "Eu-I54B3MeKSdr0XKzt5e0L0MGtoKUoCFg77ZK7EefIVL01-Tac2uJvV20FXhUGA"
}
```

*data* and the *challenge*. These values are passed as request parameters to another servlet called `FinishSign`. The *client data* is an encoded value containing the type of client used, and the challenge signed, etc. The *signature data* is also encoded, containing the value for user presence, a counter and a signature. These values are decoded at the server end, the counter value is compared to what was expected, the signature is verified using the already stored public key and then, if the signature gets verified successfully, the counter value is updated in the database.

If the client is Google Chrome, it will communicate with the token using the built-in support. However, if it's the C# handler, a host library will be needed to communicate with the key. When the handler receives the sign data, it will use the following command to pass the sign data (stored in `sign_data_file`) to the key, and will wait for the key's response too.

```
u2f-host.exe -aaauthenticate -o <app_id> < <sign_data_file>
```

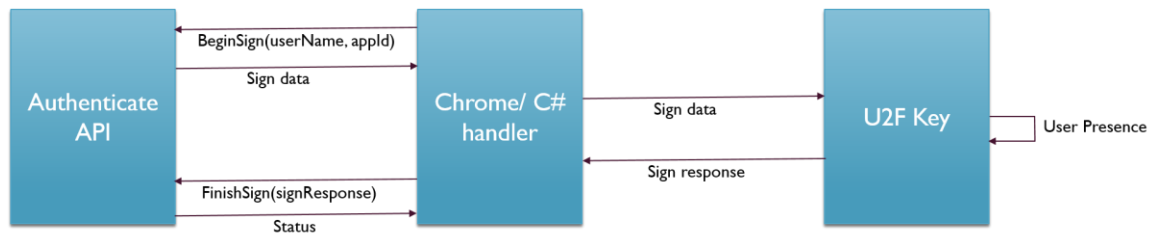


Figure 4-3: Authenticate API

## 4.2 Clients

The web APIs can be accessed using two type of clients – a web browser and a C# client.

### 4.2.1 Web Browser

Google has provided full support for U2F in its browser, mail client and OS. To access the web API, we have used Google Chrome. If you are using an older version of Google Chrome (version 39- ) or SSL is not enabled on the application server, you will require a Chrome plugin to be able to communicate with the U2F token, which can sometimes limit your mobility. In order to exploit the benefits of the built-in support in Chrome, the latest version of Chrome (version 40+) is required and SSL must be enabled on your application server. After that, it is just a matter of accessing the URL of the application and try registering and authenticating tokens. For our prototype, we have kept the UI from the open source project as it was.

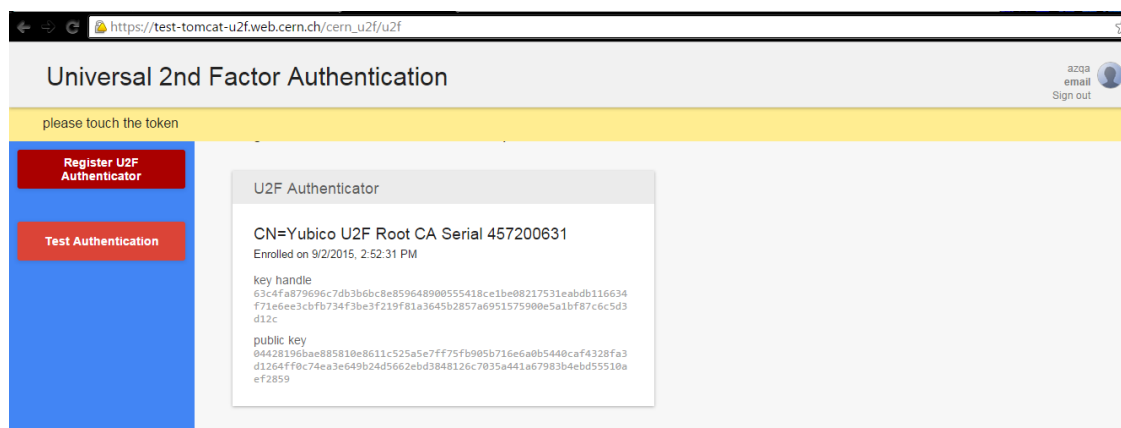


Figure 4-4: Web - Registering a token

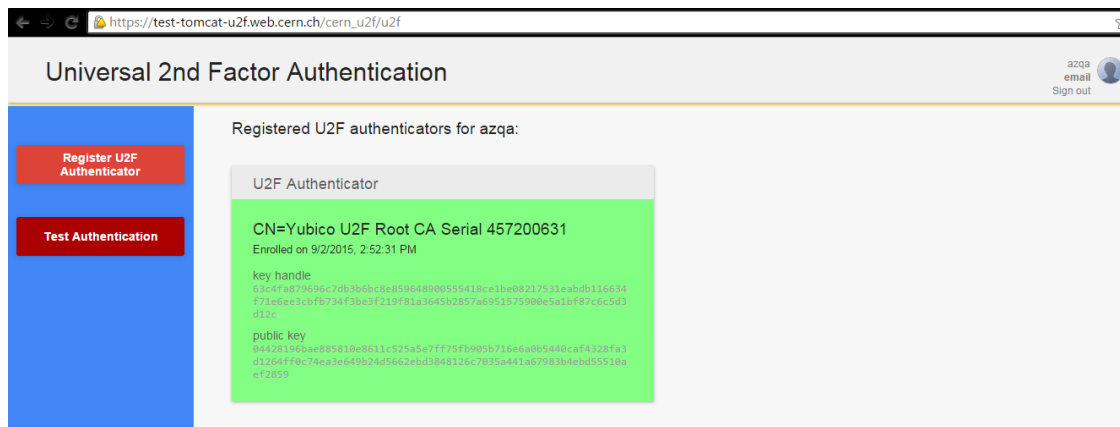


Figure 4-5: Web - Authenticating a token

## 4.2.2 C# Handler (Host Library)

CERN Self registration service is sort of like a desktop application developed in C#. Since we have to integrate the U2F module with the CERN self-registration service, we need a way to communicate with the Web APIs. Therefore, we have developed C# handlers to act as a bridge between the Web API, the Self registration service and the security keys. In order to communicate with the security keys, we also need a host library. There are two versions of host libraries developed and maintained by Yubico – one in C and the other one in Python. We have used the host library developed in C, because of its frequent releases. The library can be built on all major platforms but compiled binaries are also available. For Windows, an executable file is present in the package which we can execute using cmd.exe. So, in the C# client, after receiving the data from the server, we start a process and execute the command prompt command while passing the data received from the server, to the keys, as arguments.

There are some limitations in the host library, which are not present in the JS API that is used in the browser. For example, you can register one key more than once, and there is no way to control it, while in the JS API, you can choose if you want to re-register a token. When you register a token, a new row is created in the database for that key, and when you authenticate, the counter is incremented and the row in the database is updated. The problem with re-registration is that when you re-register a token and authenticate, only the most recently registered row is updated while the older ones just stay there as dead rows. During authentication, the server sends key handles of already registered tokens to the client. If the key finds a familiar key handle, it authenticates the request but, the library doesn't provide any option to pass the key handles along with the enrolment data, because of which the key thinks that it is new website every time and re-registers it by generating new public/private key pairs, but authenticates on the most recent pair. Another limitation was the timeout feature that the host library lacked. Whenever we have to register or authenticate a token, the library waits for an infinite amount of time for the user to press the button (to fulfil the user-presence criteria), while the JS API only waits for five seconds, after which it

times out. So, we decided to kill the process after sometime to simulate as a timeout, but fortunately the latest release of the library contains this feature, and the request times out after 15 seconds.

```

Please enter user name:
azqa
***** U2F Authenticator 1 Start *****

CN=Yubico U2F Root CA Serial 457200631
Enrolled on = 9/2/2015 2:52:31 PM
Key Handle = 63c4fa879696c7db3b6bc8e859648900555418ce1be08217531eabdb116634f71e6
ee3cbfb734f3be3f219f81a3645b2857a6951575900e5a1bf87c6c5d3d12c
Public Key = 04428196bae885810e8611c525a5e7ff75fb905b716e6a0b5440caf4328fa3d1264
ff0c74ea3e649b24d5662ebd3848126c7035a441a677983b4ebd55510aef2859

***** U2F Authenticator 1 End *****
Please enter enroll password
-- ' ' --
Please touch the token for registering
Successfully registered the token!
Registered token = {
  "enrollment_time": 1441199859125,
  "key_handle": "6075ab8eb994e89ed7107f8680bedic8d2df2e2echb1182f3a13f587c09e84
f50cd1fdec38bc605dc204b79c6a4f8d6525f2cc35ac1036cfe94ce5fb961c23",
  "public_key": "0423cea811aee39f8325becfd29a3a8db5938909305aa0172cc09fcad451b3
246369e3fd81eda7225f086fa68b19b49554e779aef052a855fbee5f5024d17c0a",
  "issuer": "CN=Yubico U2F Root CA Serial 457200631"
}

```

Figure 4-6: Handler - Registering a token

```

Please enter user name:
azqa
***** U2F Authenticator 1 Start *****

CN=Yubico U2F Root CA Serial 457200631
Enrolled on = 9/2/2015 2:52:31 PM
Key Handle = 63c4fa879696c7db3b6bc8e859648900555418ce1be08217531eabdb116634f71e6
ee3cbfb734f3be3f219f81a3645b2857a6951575900e5a1bf87c6c5d3d12c
Public Key = 04428196bae885810e8611c525a5e7ff75fb905b716e6a0b5440caf4328fa3d1264
ff0c74ea3e649b24d5662ebd3848126c7035a441a677983b4ebd55510aef2859

***** U2F Authenticator 1 End *****
***** U2F Authenticator 2 Start *****

CN=Yubico U2F Root CA Serial 457200631
Enrolled on = 9/2/2015 3:04:45 PM
Key Handle = b13d9f907058f4611c2cdeb7688eba935a957d4b517e712f88e45718919d0d461abf
18cf29a5a13914e7b6c2e1be6f9db6b930724b224a11f60906bfa462ac30
Public Key = 04e61333c29fc5dcb2bf8287319cb3bf876166e7936ea56e0da16698f8dfacdc520
8756c96529c3eb040d90fe2973c8616af12d63645fbc8b4a5bd70632be2c39c

***** U2F Authenticator 2 End *****
Please touch the token for authentication
Sign Response = {
  "signatureData": "AQAARAI0wRAIgDu263eUs3jSav_70WhhW1t1XbKDygCh9XJ1KWbr8ZQCICQk
4f6Du19kDUNKc27rif5U6C3kj0_AcHfTckBcyrdC",
  "clientData": "eyJhY2hhbGxlbmd1IjogIkFPPem9Saml2cmZaZXRyaEh0bHJaSXcILCAib3JpZ221
uIjogImh0dHBzOlw0XC90ZXN0LXRob3RhbnQhC11MmYud2UilMn1cn4uY2gilCAidHlwIjogIm5hdmlnYXR
vcisPZC5nZXRhc3M1cnRpb24iIH0=",
  "keyHandle": "Y8T6h5aWx9s7a8joWWSJAFUUGM4b41IXUx6r2xFmNPcebuPL-3NP0-PyGfgaNkWy
hXppUdZA0Wlw4fGxdPRLA"
}
Authenticated!

```

Figure 4-7: Handler - Authenticating a token

## 4.3 Persistent Storage

As previously mentioned, the server needs to remember some token information for authentication later on, e.g. the site specific public key, the corresponding key handle, counter information, etc. For that purpose, we have used Oracle Database being used at CERN currently. There is just one entity that holds information about the public key, attestation certificate, counter, key handle and a reference to which username the key belongs to.



Figure 4-8: Database Schema of the project

Token\_id is the primary key for the table, whose value is auto generated when a new row is inserted. Public\_key stores the byte encoded public key received during registration of the key. It is used to verify the signature during authentication requests. Attestation\_certificate stores a byte encoded attestation certificate of the manufacturer of the U2F token. It is used to verify if the token has been cloned or not. In our case, it was Yubico's attestation certificate. If we had encountered any other certificate, it would have been easy to verify that the token is not legitimate. Key\_handle stores the byte encoded key handle generated by the key during registration process. This is one of the parameters sent in the enrolment data, by the server to the client during authentication requests. The signature will only take place if the token finds a familiar key handle. Counter maintains the total number of times a signature attempt by the key has been taken place. A copy of the counter is also stored by the key which is incremented every time authentication or registration takes place. The purpose of this counter is to verify if an unauthorized clone of the key has been made. Lastly, every key is associated with a user account, which is referenced by the username field in the table. For demo purposes, we had created our own users table and populated it with some dummy users, but once in production, the reference of username will come from CERN's identity management server.

The configuration information of the database connection is in context.xml of the project, which is being accessed by one of the servlets to instantiate a data source connection to perform CRUD operations.

## 4.4 Deploying

After the web application was tested using the base cases, we deployed it to CERN servers. We faced some problems with the encryption library we were using. The server uses Security manager which requires a security policy file to be uploaded to be able to perform encryption and decryption on the server. After uploading the right policy file, the web application is now deployed successfully on the server. You can try it out using this URL: [https://test-tomcat-u2f.web.cern.ch/cern\\_u2f](https://test-tomcat-u2f.web.cern.ch/cern_u2f). The access for registration API has been restricted so that only CERN Self registration service providers are able to register new tokens, but you can authenticate your tokens anytime you want.



## 4.5 The Bigger Picture

We have used Eclipse (version 3.8) on Linux Mint for the development of the web API, and Visual Studio 2013 on Windows 7 Professional for the development of the C# handlers.

We started off by cloning the [implementation](#) from GitHub which is maintained by Google. We imported the project in Eclipse IDE and executed it. At this point, the server was hosted at Google App Engine, and was able to register and authenticate tokens. The only problem was, that it was storing the token information in its own data store, while we needed to store that information in CERN's database. Therefore, we created a new Dynamic Web project in Java, with Apache Tomcat 7 as the server and Oracle as the database. Then, we adapted the source code to the new requirements by modifying the data store and servlet classes. We also cleaned up the code by removing all the unneeded GAE jars and source code. To test the web APIs, we have used Google Chrome (version 41+). At this point, we were using HTTP only, so we required a [Google Chrome Extension for U2F](#) to communicate with the keys. Then, we used Nginx to enable SSL on our application, so now we are able to exploit the full benefits of the built-in support for U2F in Google Chrome along with the secure communication over HTTPS. We then tested the web APIs on localhost for different use cases.

Then, we deployed the application on CERN's Middleware on Demand test server. We faced deployment issues because of the security manager, but after uploading the right policy file, we are now able to register and authenticate U2F tokens on that website while storing token information in the respective Oracle database.

We had to develop handlers to bridge the communication between CERN self-registration service, web APIs and the security keys. For that, we required a host library to bridge the communication between the handler and the security keys. Fortunately, we found a [host library](#), developed in C, and maintained by Yubico. We have used this particular library because of its frequent releases and relatively easier setup on Windows. Because the library has been developed in C, and the handlers are developed in C#, the library can only be accessed by executing its .exe file, via the command prompt. Therefore, we start a *System.diagnostics.process* in the C# handler, execute the command from there and collect the response from the standard output and standard error. The C# handler communicates with the web APIs, receives the data, and passes it to the library to feed it to the keys. Then the library receives the response from the key, passes it to the handler, which then passes it to the web API. Please note that the data to be sent to the key is always written in a text file first. The text file is passed as an argument to the command to read the data from it because the command requires an EOF character which we cannot write in a string.



The source code for both the web application and the C# handlers is available in GitLab with operating instructions. The source code itself has also been documented with comments to help understand it well.

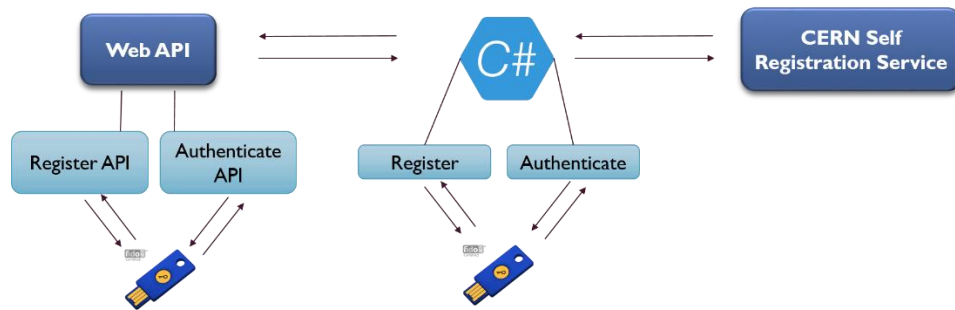


Figure 4-9: Overall architecture

## 5 Virtual Walk-through of User Interaction

This section tries to walk the reader through the user interaction with the system.

Consider a user John Doe. He wants to enable U2F authentication on his account on <https://legit.com>. He goes to the settings and select the option to do so and is asked to register a key. The browser sends a request to the registration API and server replies back with some enrolment data containing a challenge, its app id and the version of protocol the server is using. The browser asks the user to connect his key to the computer and press the button on his device. John presses the button on his device. In the meanwhile, the browser passes the enrolment data to the key. After the button is pressed, the key generates a public/private key pair and a key handle based on the user and the app id. The response is sent back to the browser which passes it to the registration web API. The server verifies the signature and stores the public key and key handle in the database. Now, whenever the user log in, the server will ask him to authenticate using the same key.

Later on, when John tries to authenticate himself into <https://legit.com>, the request is sent to the authentication API and the server replies back with some enrolment data, but this time it also contains all the key handles associated to John. The browser again asks John to connect his key to the computer and press the button. In the meanwhile, this information is passed to the key, the key tries to look for a familiar key handle and as soon as it finds one, signs the challenge with the site specific private key and responds back to the browser. The response is sent to the authentication API to verify the signature and update the counter value in the database.

For the C# client, the procedure is pretty much the same. Instead of the browser, we have a host library's executable file which performs the communication with the key along with passing requests and receiving responses from it.

We have tested the web APIs both on a workstation running on Linux, as well as one running on Windows. For Linux, you need to add a dev rule for the keys that you are mounting, and for Windows, you install the drivers for the keys which are installed automatically, the first time you connect your key to your computer. However, if you are using a key other than the standard USB device, you'll need special drivers for that.

## 6 Challenges Faced

Some of the key challenges that we faced during the project are mentioned below:

To understand the working of U2F, we needed to have some U2F enabled tokens, which we only received three weeks after the project had begun. Understanding the user interaction and control flow was a challenge without the keys. Moreover, we had to make up for the lost time.

The next challenge that we faced was regarding Google App Engine, which doesn't provide support for any other database than Google's own cloud services, while CERN uses Oracle database to store information. Therefore, we had to change the server to Tomcat so that, we could use Oracle to store token information.

The JavaScript API is more mature than the host library Yubico is providing. It was a challenge because there were some functionalities that the library lacked, while the JS API didn't. The restriction on re-registration of keys is still something that the library lacks and it might be a problem if a user wants to check if a key has already been registered or not.

Another challenge that we faced was during the deployment of the web application. The security manager in CERN servers didn't allow the application to use Bouncy Castle library, so we had to upload some security policies to get it working.

Currently, neither Microsoft nor Mozilla provide support for U2F, although they are working on it. This is a challenge because CERN uses Internet Explorer and Firefox as official web browsers, and if CERN personnel are to authenticate themselves to CERN accounts using these browsers, they will not be able to communicate with the keys. Until they provide support for U2F in these browsers, U2F as a 2<sup>nd</sup> factor cannot be deployed for production at CERN.

## 7 Conclusion

SQRL and U2F are two new 2<sup>nd</sup> factor authentication mechanisms. During the project, we have evaluated both of them and concluded that SQRL is too early in its adaption phase and has some serious security loopholes, while U2F is a more secure option with open source implementation of both server and client side being maintained by Google and Yubico.

We have implemented a web application in Java that is able to register and authenticate U2F enabled tokens. To bridge the communication between CERN Self-registration service and the web application, we have developed C# handlers that communicate with the server and the key; and are able to register and authenticate tokens.

The next step will be to provide support for Openssh and then to actually integrate the whole system with CERN SSO. After that, CERN personnel will be able to use U2F enabled tokens as another means of 2<sup>nd</sup> factor authentication.

## 8 Glossary

2 <sup>nd</sup> FA	Second factor authentication
Human presence	The security key requires human presence to make sure the user wants to register or authenticate to an application. The purpose of this flag is to avoid accidental registration/authentication
Key handle	A random value generated by the key that corresponds to the App id that the key has been registered to
Multi Factor Authentication	A mechanism for access control in which the user has to go through multiple stages to fully get authenticated to a service.
Mutual authentication	Both the server and client mutually authenticate each other
One Time Password	A random value that is valid for one time use only
SQRL	A new algorithm developed by Steve Gibson
U2F	A new algorithm for 2 <sup>nd</sup> factor authentication adapted by Google
Unilateral authentication	Only the server authenticates the client

## 9 References

- <http://community.giffgaff.com/t5/Blog/Boost-Your-Online-Security-How-To-Set-Up-Two-Factor/ba-p/16502005>
- <https://shkspr.mobi/blog/2013/09/facebook-2fa-security-flaw-disclosed/>
- [https://commons.wikimedia.org/wiki/File:CryptoCard\\_two\\_factor.jpg](https://commons.wikimedia.org/wiki/File:CryptoCard_two_factor.jpg)
- <http://lifehacker.com/5932700/please-turn-on-two-factor-authentication>
- <https://www.nexmo.com/blog/2014/11/11/why-two-factor-authentication-2fa/>
- <https://support.google.com/accounts/answer/180744?hl=en>
- <https://helpdesk.lastpass.com/multifactor-authentication-options/yubikey-authentication/>
- <http://www.authenticationworks.com/>
- <https://sqrauth.net/Logos>
- <https://commons.wikimedia.org/wiki/File:FIDO.U2F.ready.png>
- <https://www.yubico.com/applications/fido/>
- <https://www.grc.com/sqr/sqr.htm>
- <https://appengine.google.com/start>