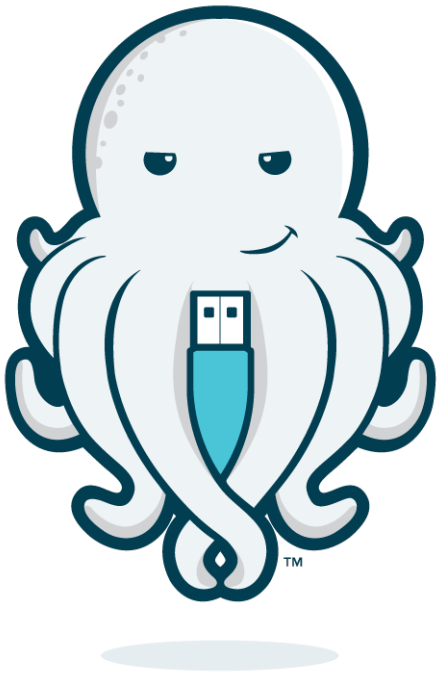


STRONGKEY

FIDO Server (SKFS)
Administration Guide

Version 4.4

Copyrights and Notices

Contents

1—Introduction

What is *Fast IDentity Online (FIDO)*? It is an industry standard authentication scheme that:

- Eliminates passwords completely from web applications
- Addresses capabilities and vulnerabilities of the worldwide web (including mobile devices)
- Is an open, royalty-free industry standard available for anyone to implement
- Incorporates user privacy as a design principle
- Is extremely simple and easy to use by anyone

This chapter provides an introduction to FIDO, providing some history and context for why a new authentication protocol was created, and what distinguishes FIDO from all previous authentication capabilities. The rest of this manual provides detailed information on how to install and use StrongKey's open-source FIDO2 server.

1.1—Background

For over two decades, the technology industry produced a plethora of *multifactor authentication (MFA)* schemes based on the principles of something you know, something you have and something you are. Ignoring the fact that the vast majority of these schemes piled additional secrets (*one-time passcodes*, a.k.a. *OTPs*; *knowledge-based answers*, a.k.a. *KBA*; or biometric templates) on top of the original secret (the password), and were, consequently, susceptible to scalable attacks similar to passwords, the vast majority of businesses and government agencies have deployed one form of MFA or the other.

Since California passed its seminal breach disclosure law in 2004, more than 10,000 data breaches were publicly disclosed compromising over 10 billion personal records. The vast majority were the result of compromised passwords. Despite the multitude of MFA solutions, they did not adequately address the problem effectively. Proprietary schemes, expensive devices, and the fact that the "gold standard" of MFA schemes—RSA's SecurID—was compromised in a scalable attack nearly a decade ago proved that secret-based authentication schemes were simply not up to the task.

What about *public key infrastructure (PKI)*? Didn't PKI break the mold by authenticating users through challenges and digital signatures without the need for a secret on the server? Indeed, it did. Many companies, government agencies, and nations invested billions of dollars in PKI. They issued hundreds of millions of smart cards, and built applications using strong authentication delivered through the *Transport Layer Security (TLS) Client Authentication (ClientAuth)* protocol enabled by X.509 digital certificates and cryptographic keys on the smart cards. However, the complexity of building and maintaining a PKI, and the challenging *user experience (UX)* when working with digital certificates left a lot to be desired.

Around 2014, a group of companies believed the time had come for something stronger and better to eliminate passwords off the internet. The world of computing and the worldwide web had changed dramatically and something different was needed. A

non-profit standards group, the FIDO Alliance was created, and with over 200 companies worldwide as members—including the governments of US, UK, Australia, Germany—2019 saw the latest version of the FIDO protocol standardized by the FIDO Alliance, with the *World Wide Web Consortium (W3C)* supporting a standard JavaScript *application programming interface (API)*—WebAuthn—to define a standard way of integrating FIDO into web applications. Some of the world's largest internet sites, corporations, and the US government have done so with many more coming. Every modern web browser (with the exception of the legacy Internet Explorer), Windows 10, Android (7 or greater), and iOS currently supports FIDO without the need for special drivers, middleware, or readers.

1.2—Differences

What distinguishes FIDO from every other authentication protocol from the past?

Like TLS Client Authentication, it uses public-key cryptography to eliminate secrets on the server. But, unlike the X.509 digital certificate that underlies TLS ClientAuth, FIDO does not need a PKI. As a result the cost, complexity and challenging UX of using smart cards and digital certificates are obviated.

Like MFA schemes that use a hardware authenticator, FIDO uses cryptographic hardware modules that may be embedded within devices—such as the *Trusted Platform Module (TPM)* on personal computers, or *Secure Elements (SE)* on Android phones, or external authenticators called Security Keys in FIDO terminology. These hardware elements are designed to protect cryptographic keys to mitigate scalable attacks on keys.

FIDO is unique in including the following features:

- FIDO requires the generation of a new, unique key pair (public key and private key) for every website with a unique web origin consisting of a *top-level domain (TLD)* (such as .com, .net, .org) plus one sub-domain (such as *strongkey*). As a result, FIDO Authenticators generate unique key pairs for a user's credential at strongkey.com. Every FIDO-enabled website is required to declare a relying party identifier (a.k.a. *rpid*). While a relying party (RP)—the company site that relies upon a FIDO digital signature to authenticate a user's credential—may have many websites (*register.strongkey.com*, *support.strongkey.com, login.strongkey.com*, etc.), they may use the same *rpid* (*strongkey.com*) and FIDO server to support a single FIDO credential per user across all its sites. This enables authenticating the user to each site separately, providing the advantage of distributing authentication workloads across a cluster of FIDO servers while eliminating the burden of adding a legacy protocol to web applications to support *single sign-on (SSO)*. FIDO mandates the use of TLS, so all communications on the wire are secure. However, FIDO assumes that the client platform, usually the browser, is secure—FIDO cannot protect a user from a compromised platform.
- FIDO uses two distinct protocols to communicate between the RP and the Authenticator: the W3C JavaScript API—WebAuthn—between the RP web application and the client platform—usually the browsers; and the FIDO Alliance's *Client To Authenticator Protocol (CTAP)* between the client platform and the Authenticator. Since both WebAuthn and CTAP are standard protocols, users as well as RPs are freed from the burden of attempting to make a complex set of software and hardware components work together on a multitude of operating systems and devices; each platform vendor—whether it is Microsoft, Google, Mozilla, Apple, etc.—assumes the responsibility of making sure that WebAuthn and CTAP work precisely as expected on their platforms.

- FIDO allows the registration of any number of unique Authenticators to an *rpid*. This is unlike password technology that uses only one username and a single password. While X.509 digital certificates permit issuing unique digital certificates to the same user and allowing the user to use any of the set of digital certificates to authenticate to a TLS ClientAuth-enabled site, this is rarely done in practice. However, FIDO allows a user to use the TPM on a Windows 10 personal computer, the SE on an Android phone, the SE accessed through TouchID on a MacBook, and an external Security Key to be all registered to the same username at a specific site and use any of these four authenticators to authenticate to the site. This has the advantage of eliminating the problem of account recovery: how to gain access to your account if you lose your primary authenticator. The user simply uses another authenticator registered with the site, authenticates to the site, goes into their profile settings, deletes the lost authenticator's registered key, and carries on. They may choose to replace the lost authenticator with a new one, but as long as a user has at least two keys registered with a site, they are unlikely to be locked out of their account. Even if both authenticators are lost/unusable, most RP sites will have a non-FIDO-based account recovery process which is likely to fall back to sending an e-mail reset link plus an OTP to your mobile, etc., to gain access to the site. Once authenticated, the user may register any number of new authenticators to their account again.
- FIDO eliminates the risks associated with password phishing. Cleverly crafted spear-phishing attacks to spurious websites that prompt for usernames/passwords are pointless if the authentic site supports FIDO-based passwordless authentication; no password is provided to the attacker's site. Additionally, a man-in-the-middle (MITM) attack that attempts to relay a user's authentication session through the attack site will fail since the attacker's site will have a different *domain name system (DNS)*-based web origin. FIDO platforms such as browsers or *rich-client apps (RCA)* on mobile devices are required to map the RFC-6454v web origin to the *rpid* and ensure they have the same TLD+1 before using the private key to digitally sign a challenge.
- FIDO mandates a *test of user presence (TUP*) that requires an authenticator to verify the presence of a human being at the client device attempting to authenticate to a site. While the FIDO protocol mandates the TUP, it leaves the TUP's implementation up to manufacturers of authenticators so the market may innovate and provide a variety of solutions. Some external authenticators merely require you to touch a metallic sensor (generally, with a blinking light-emitting diode) on the device to prove TUP. While this implementation does not identify a specific user, it provides the RP evidence of possession of the right authenticator containing a private key corresponding to the registered public key at the RP site, as well as the presence of a human being at the client device with that authenticator. The design of the FIDO protocol takes into account that platforms prevent attackers from gaining access to lower-level communication protocols—such as the *universal serial bus (USB)*, *near-field communications (NFC)* or the *Bluetooth low energy (BLE)* transport protocols—between the platform and an external authenticator.
- Where RP sites require the authenticator to verify the identity of the user at the client device, FIDO supports the use of embedded device authentication capabilities to support *user verification (UV)*. An external authenticator with biometric capabilities; the MacBook with TouchID; a Windows 10 personal computer with a fingerprint reader; a mobile device with a fingerprint reader, *personal identification number (PIN)*, or a geometric pattern reader—any of these provide assurance to the RP application that the FIDO authenticator verified the identity of the user before authorizing the use of a

private key to digitally sign the FIDO challenge. In keeping with privacy principles of the FIDO protocol, neither the biometric template, nor the PIN/pattern are shared with the RP site—they are used only to verify the user identity locally on the authenticator device/platform, and when verified, unlock the private key to sign the FIDO challenge.

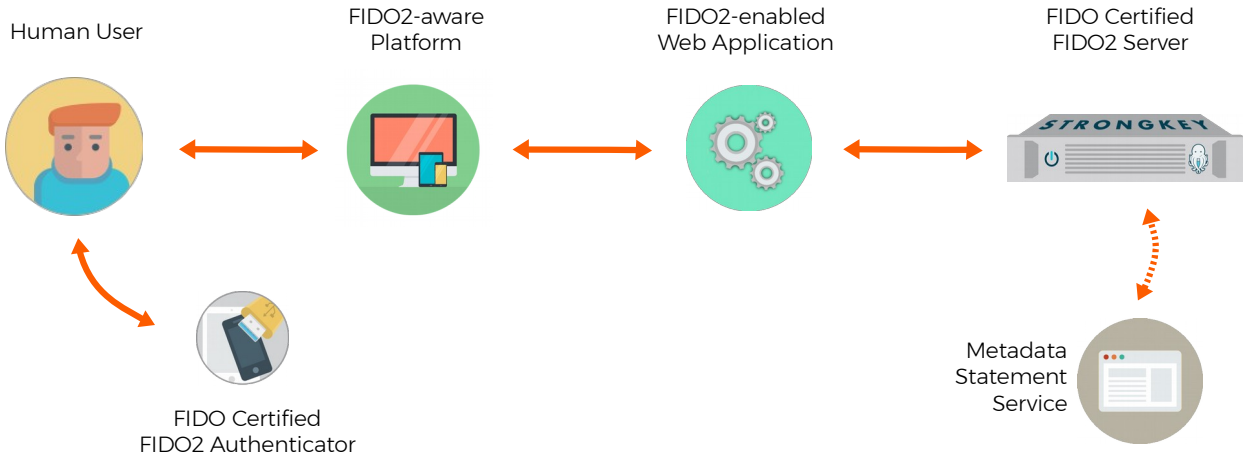
- FIDO protocol can require that assertions—digital signatures generated during the FIDO authentication process—cannot be replayed by requiring the FIDO server to send a unique nonce (monotonically incrementing number used once) as part of the challenge, and having the authenticator sign the nonce with the private key. The FIDO server not only verifies the digital signature in the assertion, but also verifies the accuracy of the nonce with what the server sent earlier, and that the nonce was not used in any prior authentication.
- With the appropriate platform and authenticator, FIDO has the potential to deliver a frictionless user experience for a business transaction. For instance, a transaction performed on a device with a fingerprint reader could render transaction details to the user and prompt for a FIDO digital signature asserting to the transaction details. This obviates the need to send any *short message service (SMS)*-based OTP to the user's mobile phone or to e-mail an OTP. This is also true of mobile devices. This frictionless experience can have a profoundly positive experience with users while drastically reducing risk for an RP due to the security of the FIDO protocol.

FIDO is one of the most innovative authentication solutions that has appeared on the landscape in a quarter of a century. NIST declared FIDO to provide the "highest assurance level" for an authentication protocol in its Special Publication 800-63vi. While it still remains the responsibility of users and/or RPs to ensure that Authenticators are secure enough for their purposes, FIDO Alliance has defined a security certification process conceptually analogous to the *Federal Information Processing Standards (FIPS)* 140-2, but whose tests are specific to features defined within the FIDO protocol. Manufacturers of Authenticators may choose to submit their devices to independent laboratories to test them for specific security levels, then provide the FIDO Alliance results for certification. With configurable FIDO servers, RPs may define security policies that are suitable for their business purposes.

There are dozens of implementations of FIDO Certified® Authenticators and servers, including StrongKey's open-source FIDO Certified® server. If you have a recent build of Windows 10 on your computer, a MacBook with TouchID, an Android 7 (or greater) phone, or an iPhone with the latest build of Safari, we encourage you to try the experience of registering and using a FIDO authenticator at sites where they are currently supported—FIDO is certain to play a central role in your security defense strategy in the coming decade.

1.3—Architecture

A simplified high-level architecture of a FIDO web application—without showing any *high availability (HA)* or *disaster recovery (DR)* infrastructure—resembles the following diagram:



1.3.1—Architecture Notes

- FIDO Authenticators come in many shapes and forms; they may be embedded as a chip within a desktop, laptop, tablet computer or a mobile phone, or be a discrete component outside the computing device (Security Key), but connected to it over a USB, BLE or an NFC connection
- A FIDO-aware platform may be a browser, an operating system, or even a rich-client application that supports the WebAuthn API and CTAP
- A FIDO-enabled web application may use any programming language or framework as long as they support the required API and protocols to transmit messages between the FIDO Server and the Authenticator
- The FIDO Server may be embedded as a library within the web application back end or may be a distinct FIDO Server providing services to any number of applications within an enterprise (as is shown in this diagram)
- The *Metadata Statement (MDS)* Service is a FIDO Alliance capability to aid Relying Parties in learning specific Authenticator characteristics for risk-management decisions; this is an optional—but recommended—component of the architecture, as it allows RPs to learn of unknown vulnerabilities or recalls within specific Authenticators used by its applications

As with all StrongKey products, the *StrongKey FIDO2 Server (SKFS)* is a *Java Enterprise Edition (JEE)* application. It relies upon the following open-source components:

- CentOS 7.9, Ubuntu 18.04, Debian 9
- Open Java Development Kit (OpenJDK) 1.8
- Payara Application Server 5
- MariaDB Relational Database Management System (RDBMS) 10.5.80
- OpenDJ 3.0
- BouncyCastle FIPS 1.0.1

1.3.2—Capabilities

SKFS supports the following capabilities:

- Register and authenticate *Universal 2nd Factor (U2F)* and FIDO2 keys from FIDO Certified® Authenticators
- Register, Authenticate, and Authorize (in support of transaction confirmation with a digital signature) when using a mobile app built using the *StrongKey Android Client Library (SACL)*
- Accessing SKFS capabilities through *Representational State Transfer (REST)* or *Simple Object Access Protocol (SOAP)* web services from web/mobile applications. It is important to recognize that the client application—whether executing within a mobile device or in a browser on any computing platform—does not interact with SKFS directly; all FIDO interactions are intended to be available through the application's back-end services
- SKFS deploys in a variety of configurations to address customer risk management requirements:
  - Leveraging the StrongKey Tellaro appliance with a FIPS 140-2 Level-2 or Level-3 cryptographic hardware module (to offer the highest level of security), in a single-tenant deployment either on customer premises or in the StrongKey Cloud
  - Leveraging the StrongKey Tellaro appliance with a FIPS 140-2 Level-2 or Level-3 cryptographic hardware module (to offer high levels of security), in a multi-tenant deployment in the StrongKey Cloud
  - As a software-only deployment within a Docker container in private or public clouds —please review the appendix on the security risks of deploying cryptographic key management systems in a multi-tenant public cloud
- HA/DR through the deployment of multiple SKFS instances within a cluster of peer nodes. Each node in the cluster supports all operations of SKFS and asynchronously replicates its objects to all other peer nodes of the cluster across a local or wide area network
- *Lightweight directory access protocol (LDAP)* directory server—such as OpenDJ or *Active Directory (AD)*—integration to enable registering existing users with their FIDO keys after they are authenticated to the Directory Server
- Public key infrastructure (PKI) integration to enable registering existing users with their FIDO keys after they are strongly authenticated with their X.509 digital certificates;
- Digitally signing all FIDO objects upon persistence and verifying object digital signatures upon read to assert the integrity of FIDO objects—this prevents attackers from substituting public key handles of users within a FIDO database to attack other users' accounts without the need for their FIDO Authenticator
- Site and application-specific policy management to tailor SKFS deployments to customers' risk management profiles
- Bundled software FIDO Authenticator to support testing web/mobile applications within *continuous integration (CI)* and *continuous deployment (CD)* environments
- Bundled *command line interface (CLI)* tool to test SKFS deployment and configuration
- Bundled JMeter plan to support customers testing their web/mobile applications for performance bottlenecks and HA/DR capabilities

All features referenced in this manual are available with StrongKey support—whether SKFS is downloaded from public repositories such as GitHub or SourceForge, or acquired through the StrongKey Cloud or StrongKey Tellaro appliances; the only difference in support is that a contract with StrongKey guarantees a *service level agreement (SLA)* the on which the customer can depend for a response to the support request. StrongKey supports all other uses of SKFS through the forums on GitHub and SourceForge on a best-efforts basis.

2—SKFS Installation & Upgrade

2.1—Standalone

2.1.1—Prerequisites

- One of the following Linux distributions; the installation script is untested on other flavors of Linux but may work with slight modifications:
  - RedHat/CentOS/Oracle 7
  - Ubuntu 18.04
  - Debian 9
  - Amazon Linux 2
- A *Virtual Machine (VM)* with a minimum of 20GB space and 4GB memory assigned to it; some default VMs do not allocate sufficient space and memory, so please verify before getting started
- A public *fully qualified domain name (FQDN)*. It is very important to have a hostname that is at least *top-level domain (TLD)*+1 (i.e., acme.com, example.org, etc.); otherwise FIDO2 functionality may not work
- The installation script installs Payara running HTTPS on port 8181, so make sure all firewall rules allow that port to be accessed
- *StrongKey FIDO2 Server (SKFS)* must be installed before the sample service provider web application and sample WebAuthn Java client.

2.1.2—Installation

NOTE: If the install fails for any reason, follow the instructions for Removal, below, and restart from the beginning.

1. Open a terminal and change directory to the target download folder.
2. Install *wget* if it has not been already.

   ```
   shell> sudo yum install wget
   ```

   or

   ```
   shell> sudo apt install wget
   ```

3. Download the binary distribution file `fido2server-v4.3.0-dist.tgz`.

   ```
   shell> wget https://github.com/StrongKey/fido2/raw/master/fido2server-
   v4.3.0-dist.tgz
   ```

   Extract the downloaded file to the current directory:

   ```
   shell> tar xvzf fido2server-v4.3.0-dist.tgz
   ```

4. Be sure the machine's **FQDN is set as its hostname**. This is necessary to properly configure the self-signed certificate for the API. Verify using the following command:

```
shell> hostname
```

If only the machine name is returned, and not the public FQDN, run the following command:

```
shell> sudo hostnamectl set-hostname <SERVER PUBLIC FQDN>
```

If no DNS is configured for this machine, please run the following command to add an entry to the `/etc/hosts` file. **DO NOT run this if the machine does not have a configured FQDN and is still running as *localhost*.**

```
shell> echo `hostname -I | awk '{print $1}'` $(hostname) | sudo tee -a
/etc/hosts
```

5. **Execute** the `install-skfs.sh` script as follows:

NOTE: If you are installing on Ubuntu VM, please make sure you are using *bash* as your default. If the default is set to *sh*, please execute `sudo dpkg-reconfigure dash` to change the default to *bash* before continuing.

```
shell> sudo ./install-skfs.sh
```

The installation script will create a *strongkey* user account with the home directory of `/usr/local/strongkey`. All software required for SKFS will be deployed to the `/usr/local/strongkey` directory and be run by the *strongkey* user. The default password for the *strongkey* user is *ShaZam123*.

6. Using the following command, **confirm SKFS is running**. The API *Web Application Definition Language (WADL)* file comes back in response.

```
shell> curl -k https://localhost:8181/skfs/rest/application.wadl
```

7. To **test this installation** of SKFS, check out the Basic Java Sample application or a JAVA *proof of concept (PoC)* application, which also involves user registration via emails.

NOTE: Both the signing and secret keys in the keystore use default values and should be changed after installation is completed. The keystore and the TrustStore are located in `/usr/local/strongkey/skfs/keystores`. Run the following command from `usr/local/strongkey/keymanager` to see the usage and syntax for the *keymanager* tool, then change them both (The default password for the files is *Abcd1234*!):

```
shell> java -jar keymanager.jar
```

2.1.3—Removal

To uninstall StrongKey FIDO2 Server, run the following command from the folder where the distribution was extracted:

```
shell> sudo ./cleanup.sh
```

This removes all StrongKey files plus the installed dependency packages. If the sample service provider web application and the StrongKey WebAuthn Java client are installed, they will be removed as well.

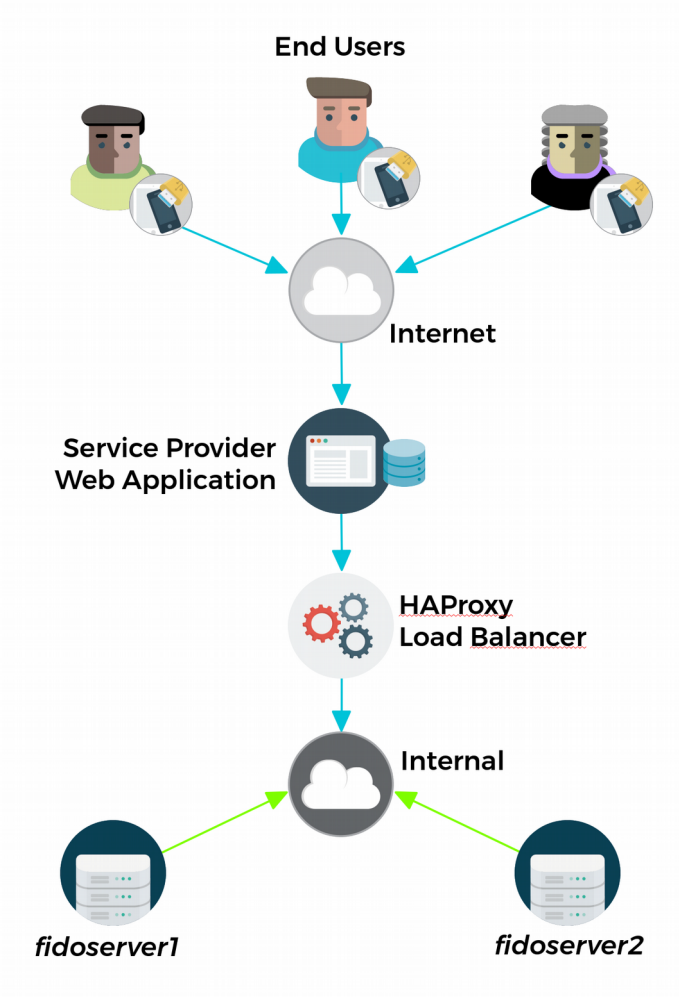
2.2—Clustered Installation

SKFS can be clustered with multiple nodes to deliver *high availability (HA)* across a *local area network (LAN)* and/or *disaster recovery (DR)* on a *wide area network (WAN)*. No additional software is required to enable these features because StrongKey has enabled this capability as a standard feature in its FIDO2 server. Furthermore, with multiple nodes processing FIDO2 transactions at the same time, the SKFS cluster can deliver higher throughput to multiple web applications that use this server. This document guides you through the setup of a cluster with two nodes, as depicted in the image below.

The clustering capability in SKFS only applies to the FIDO2 capability. Web applications that use SKFS must make their own arrangements to deliver HA and/or DR independent of SKFS. The sample application used here to demonstrate FIDO2 clustering will, itself, not be highly available, but demonstrates that the web application can use either or both FIDO2 servers in this HA configuration.

While it is possible to add more than two nodes to the cluster, IT architects will recognize that there is a trade-off with N-way replication designs—the more nodes, the higher the resource requirements on each node to manage fail-safe replication, which can reduce the overall throughput after a certain point. Each site will have to do its own testing to discern where the throughput curve flattens out. However, if you have truly large-scale deployments in mind, please contact us to see how we can help.

2.2.1—Sample Cluster Configuration



2.2.2—Prerequisites

1. Two virtual machines (VMs) for the FIDO2 servers, running the current version of CentOS Linux 7.x, with FQDN and internet protocol IP addresses.
2. One virtual machine for the load balancer, running HAProxy version 1.5.18 software on the current version of CentOS Linux 7.x with an FQDN and an IP address
3. One virtual machine for the StrongKey sample Proof-of-Concept web application from this GitHub repository, also running on the current version of CentOS Linux 7.x with an FQDN and an IP address.

NOTE: This document assumes you are setting up this cluster with all nodes connected to a single Ethernet switch. If your intent is to do a more realistic test, you should plan on using VMs with multiple network interfaces connected to different switches to isolate traffic to the appropriate segments, as you might expect in a real-world environment.

2.2.3—Cluster Setup

Using the installation steps here, install and configure the two FIDO2 server VMs as if they were individual FIDO2 servers, but do NOT install any web applications to test the FIDO2 server at this point; we will do this later.

For each server determine the FQDN and assign it a unique Server ID. A *Server ID (SID)* is a numeric value that uniquely identifies a node within the cluster. Conventionally, StrongKey cluster SIDs begin with the numeral 1 and continue incrementally for each node in the cluster.

- **SID 1** is fidoserver1.strongkey.com
- **SID 2** is fidoserver2.strongkey.com

1. As the *root* user, perform the following sub-tasks on every node to be in the cluster:
   a. **Login** as *root*.
   b. **If DNS is configured**, make sure it is configured for **forward and reverse lookups**—meaning that it should be possible to resolve the IP address using the FQDN, as well as resolve the FQDN using the IP address doing a reverse lookup. Without the reverse resolution, services in the Payara application server configuration will not work correctly.

      If Domain Name Service (DNS) is not configured, add the following entries to the `/etc/hosts` file to identify the cluster nodes. Use a text editor such as *vi* to modify the `/etc/hosts` file. For the two-node cluster, add the following to the end of the `hosts` file, substituting the *strongkey.com* domain name for your own environment:

      ```
      <ip-fidoserver1>     fidoserver1.strongkey.com fidoserver1
      <ip-fidoserver2>     fidoserver2.strongkey.com fidoserver2
      ```

   c. **Modify** the firewall configuration to open ports 7001, 7002, and 7003 to accept connections between just the FIDO2 servers to enable multi-way replication. Run the following command once for each cluster node's IP address (substituting for `<ip-target-fidoserver>`).

      Do not execute this command for the IP address of the cluster node on which you are executing the command itself. It is not necessary to open the node's ports on the firewall for itself, since the replication module in SKFS does not need to replicate to itself.

      ```
      shell> firewall-cmd --permanent --add-rich-rule 'rule family="ipv4"
      source address='<ip-target-fidoserver>' port port=7001-7003
      protocol=tcp accept'
      ```

   d. After adding the new rule, **restart the firewall**:

      ```
      shell> systemctl restart firewalld
      ```

   e. **Logout** from the *root* account.

2. As the *strongkey* user (the default password is *ShaZam123*), perform the following on every SKFS node to be clustered:
   a. Using a text editor, **edit the configuration properties of the SKFS node**; if the specified file is empty add these properties:

```
shell> vi /usr/local/strongkey/appliance/etc/appliance-
configuration.properties
appliance.cfg.property.serverid=<server-id> (set the value to the
corresponding SID of the current node)
appliance.cfg.property.replicate=true (should be set to true)
```

   b. Using the MySQL client, **login to the MariaDB database** that was installed as part of the SKFS installation. The default password for the *skfsdbuser* is *AbracaDabra*.

```
shell> mysql -u skfsdbuser -p skfs
```

   c. **Truncate** the existing SERVERS table—this deletes all contents of the SERVERS table:

```
mysql> truncate SERVERS;
```

   d. **Insert the following entries** into the SERVERS table, ensuring the SID and FQDN match the values used in *Step 2*. For example, if there are two nodes in the cluster, add the following entries:

```
mysql> insert into SERVERS values (1, 'fidoserver1.strongkey.com',
'Active', 'Both', 'Active', null, null);
mysql> insert into SERVERS values (2, 'fidoserver2.strongkey.com',
'Active', 'Both', 'Active', null, null);
```

   e. **Logout** of the MySQL client:

```
mysql> exit
```

   f. **Import the self-signed certificates** generated as part of the FIDO2 server installation into the Payara Application Server's truststore—this is necessary to ensure that replication between the FIDO2 server nodes occurs over a trusted *Transport Layer Security (TLS)* connection. Execute the `certimport.sh` script included in the `/usr/local/strongkey/bin` directory to import the certificate.

```
shell> /usr/local/strongkey/bin/certimport.sh fidoserver1.strongkey.com
-kGLASSFISH
shell> /usr/local/strongkey/bin/certimport.sh fidoserver2.strongkey.com
-kGLASSFISH
```

   g. **Restart** the Payara Application Server (even though we refer to it as the Payara Application Server, the startup script is named `glassfishd` for legacy reasons, given Payara's origins from the open-source GlassFish Application Server):

```
shell> sudo service glassfishd restart
```

   h. **Repeat** *Steps 3 and 4* on the remaining SKFS nodes of this cluster.

2.3—Installing Dockerized SKFS

The */docker* branch contains a Dockerized version of StrongKey's Certified FIDO2 Server, Community Edition. This implementation allows for the creation of a FIDO2 server within a container, which allows for the ability to be deployed in any environment. This section will focus solely on the Docker portion of SKFS.

2.3.1—Prerequisites

2.3.1.1—Host machine/VM

Docker must be installed and enabled to build and use the SKFS image. We recommend you have a machine or VM with the following minimum requirements:

- 10 GB storage
- 4 GB memory

These values are also the recommended minimums for each container. CPUs and memory for each container may be manually allocated at runtime (see example usage). If these flags are not set, "by default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows."

The host machine's firewall should open port 8181 (or the port bound to the Docker container's port 8181) as well as ports 7001–7003 if clustering containers.

2.3.1.2—External Database and LDAP/AD

In this specific version of the Dockerized SKFS, both the MySQL database and Lightweight Directory Access Protocol (LDAP) are not included, so an external MySQL 5 database and LDAP/Active Directory (AD) are required for setup. The containers can be configured to use the external database and LDAP/AD in the `base64-input.sh` script.

The database configuration should be sourced from `/fidoserverSQL/create.txt` on the server. Additionally, any database insert commands found in `install-skfs.sh` from a non-Dockerized SKFS should be performed on this database.

The external LDAP/AD must be configured in the same way as a normal FIDO2 Server as described in the distribution's `.ldif` files.

2.3.2—Dockerize!

1. Build the image.
   ```
   [sudo] docker build -t fidoserver .
   ```
2. Configure the image settings. Here you may configure clustering and the external LDAP/AD database:
   ```
   vi base64-input.sh
   ```
3. Get base64 input for the container. Save the output of the `base64-input.sh` script somewhere for the next step.
   ```
   ./base64-input.sh
   ```

4. **Run** the image in a container with the output from the `base64-input.sh` script as its only argument. The hostname flag (`-h`) and bound port 8181 are necessary. To debug any issues with GlassFish (Payara), it may be a good idea to remove the `-d` (`detach`) flag to be able to view the logs

   ```
   [sudo] docker run -dit -h fido01.strongkey.com -p 8181:8181 fidoserver
   <base64-input>
   ```

To enter into a bash terminal within the container, perform the following additional steps:

5. **Find** the *container ID*.

   ```
   [sudo] docker container ls
   ```

6. To enter into a bash terminal in the container, **execute the following**:

   ```
   [sudo] docker exec -it <CONTAINER-ID> /bin/bash
   ```

2.3.3—Clustering

When clustering, for replication to work properly, open ports 7001–7003 on the container when running it. For example:

```
[sudo] docker run -dit -h fido01.strongkey.com -p 8181:8181 -p 7001-
7003:7001-7003 fidoserver <base64-input>
```

Further instructions on clustering can be found in `base64-input.sh` when configuring the image settings on *Step 2* of the *Getting Started* section.

2.4—Install HAProxy Load Balancer

HA is enabled for applications by inserting a load balancer between components of the infrastructure, such as between the web application and the two FIDO2 servers of this configuration. The load balancer determines which target server is available to receive application connections, and distributes application requests to the appropriate one.

SKFS has been tested with the open-source HAProxy load balancer, part of the standard CentOS Linux distribution. It is conceivable that SKFS will work with other load balancers; please contact us to discuss your needs.

To install and configure HAProxy for use with the FIDO2 server cluster, follow the steps below:

1. **Install** the standard CentOS 7.x Linux distribution on one of the four VMs provisioned for this setup.
2. **Login** to the server as *root*.
3. **Install HAProxy** using the Yellowdog Updater, Modified (*yum*) tool:

   ```
   shell> yum install haproxy
   ```

4. **Create a self-signed certificate** to be used by HAProxy, replacing the value in the *-subj* parameter with the value relevant to your site. The most important element within this parameter is the *CN* component—the value must match the FQDN of the VM used for this load balancer; so if you choose to name your VM *fidoserver.mydomain.com* then the *-subj* parameter may simply be */CN=fidoserver.mydomain.com*:

```
shell> openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/etc/pki/tls/private/fidoserver.key -out
/etc/pki/tls/certs/fidoserver.crt -subj "/CN=saka02.strongkey.com"
```

5. Concatenate the generated key and certificate files into a single file, preserving the names of the files as shown below:

```
shell> cat /etc/pki/tls/certs/fidoserver.crt
/etc/pki/tls/private/fidoserver.key > /etc/pki/tls/certs/fidoserver.pem
```

6. Using a text editor, edit the HAProxy configuration file to make the following changes:

```
shell> vi /etc/haproxy/haproxy.cfg
```

7. Replace the contents with the following and replace the `<ip-fidoserver1>` and `<ip-fidoserver2>` parameters with the IP addresses for the FIDO2 servers:

```
global
    log 127.0.0.1   local0
    log 127.0.0.1   local1 debug
    maxconn   45000 # Total Max Connections.
    daemon
    nbproc      1 # Number of processing cores.
defaults
    timeout server 86400000
    timeout connect 86400000
    timeout client 86400000
    timeout queue   1000s

listen  https_web
    bind *:443 ssl crt /etc/pki/tls/certs/fidoserver.pem
    option tcplog
    mode http
    balance roundrobin
    option forwardfor
    server server1 <ip-fidoserver1>:8181 check ssl verify none
    server server2 <ip-fidoserver2>:8181 check ssl verify none
```

8. **Create a firewall rule** to open port **443** to allow the web application to communicate with the load balancer:

```
shell> firewall-cmd --permanent --add-rich-rule 'rule family="ipv4" port
port=443 protocol=tcp accept'
```

9. **Restart HAProxy**:

```
shell> service haproxy restart
```

10. Verify HAProxy is functioning as expected by **accessing the URL in the browser**. If it is functioning correctly, it will redirect you to one of the configured FIDO2 servers.

```
https://<fidoserver.mydomain.com_>
```

2.4.1—*selinux*

If you cannot access the above URL in the browser, ensure that the *selinux* config has been set to permissive instead of enforcing.

The following command will show you the current status of *selinux*:

```
shell> sestatus
```

If it is set to enforcing, change it to permissive by running the following command:

```
shell> setenforce 0
```

2.5—Simulating Node Failures in the SKFS Cluster

Following are several methods to simulate failures of a node within the cluster for verification purposes:

- Remove the Ethernet cable from one of the SKFS nodes
- Shut down the Payara Application Server on one of the FIDO2 server nodes
- Close port 8181 by disabling the firewall rule that accepts connections on SKFS
- Modify the configuration of HAProxy on the load balancer to remove one FIDO2 server

NOTE: Because of the complexity of the FIDO2 protocols as well as its implementation in SKFS, some in-flight FIDO2 registrations and/or authentications may see failures due to the simulated outage (as might occur in a real-world environment). Application architects should consider how they might choose to address these failures within their web applications—the PoC was designed to demonstrate simple FIDO2 transactions and not a specific application.

StrongKey definitely appreciates feedback on how it might improve the FIDO2 server to better serve this community's needs. Please share your ideas on the forum on GitHub.

2.6—Removal

To uninstall StrongKey FIDO2 Server, run the following command on every server in the cluster from the folder where the distribution was extracted:

```
shell> sudo ./cleanup.sh
```

This removes all StrongKey files and any installed dependency packages. If the sample service provider web application and the StrongKey WebAuthn Java client are installed, they will be removed as well.

2.7—Performing a StrongKey FIDO2 Server Upgrade

For older versions of the StrongKey FIDO2 Server, upgrading to the current version (4.4) of the FIDO Server is possible through the use of the upgrade script. To upgrade, simply run the `upgrade.sh` script as the *root* user. A database backup is created during the script, but it is recommended to make another database backup with the command:

```
$> /usr/local/strongkey/mariadb-10.2.30/bin/mysqldump -u root -p skfs >
skfs_backup.sql#
```

2.7.1—Upgrade Script Configurability

Editing the upgrade script allows the user to configure details new updates or whether to upgrade to newer major versions of Payara or MariaDB.

The upgrade script configurables primarily include values added to the updated FIDO2 Server policy in newer updates. The following shows the configurables with brief explanations of each group: The `RPNAME` and `RPID` flags change the relying party id and name in the policy to be added to the `fido_policies` table in the database.

```
RPNAME=FIDOServer
RPID=strongkey.com
```

The following flags are used for both the new FIDO policy to be added in the database as well as inputs for the new JWT key generation script.

```
JWT_DN='CN=StrongKey KeyAppliance,O=StrongKey'
JWT_DURATION=30
JWT_KEYGEN_DN='/C=US/ST=California/L=Cupertito/O=StrongAuth/OU=Engineering'
JWT_CERTS_PER_SERVER=3
JWT_KEYSTORE_PASS=Abcd1234!
JWT_KEY_VALIDITY=365
SAKA_DID=1
```

As mentioned earlier, flags have been added to let the user decide whether to upgrade to a newer major version of Payara or MariaDB. These flags are set to update both by default.

```
UPDATE_GLASSFISH=Y
UPDATE_MARIADB=Y
```

The `ROLLBACK` flag maintains previous Payara and MariaDB versions by default. If set to 'N', the old versions will be deleted.

```
ROLLBACK=Y
```

2.7.2—Additional Notes

Post-upgrade, if the previous 4.3.0 version added new users to LDAP, those users will need to be added to the new LDAP groups:

- `FidoRegAuthorized`
- `FidoSignAuthorized`
- `FidoAuthzAuthorized`
- `FidoAdminAuthorized`

If SKFS is in a cluster configuration, the generated `jwtsigningkeystore.bcfks` and `jwtsigningtruststore.bcfks` must be copied to the other SKFS instances.

3—Administration

3.1—Operations

SKFS can be installed using one of two methods:

- Standalone configuration using downloaded code from GitHub
- If you purchased a Tellaro appliance, follow the instructions in the *KA Administration Guide, Chapter 4—Installation*

3.1.1—Deployment Considerations

FIDO2 Server only exposes web services: *Representational state transfer (REST)* and *Simple Object Access Protocol (SOAP)*. The server does not have to be built into web applications, and can run as an independent instance enabling all applications—regardless of the language/framework with which they're built—to use FIDO2 capabilities.

While the Tellaro appliance's cryptographic hardware module provides a tamper-evident key management device and enjoys all the benefits therefrom, the FIDO2 Server demo running in a *Virtual Machine (VM)* is not guaranteed to have the same level of security. When deployed on a VM, the secure element is simulated by software, which is by default not as secure as the hardware equivalent. For this reason StrongKey recommends not deploying the FIDO2 Server for a production environment in a VM, but instead only as a proof of concept for demonstration purposes.

When considering FIDO2 server deployment options, one can use a FIDO2 library and add it to the web application or deploy an independent server, as available on the StrongKey Tellaro. The independent server in the Tellaro has many advantages over libraries:

- It makes the web application “lighter” since it does not link any non-core libraries into the business application, and enables the application to merely make web service requests to use FIDO2 services
- It enables multiple web applications to use common, clustered FIDO2 Servers to strongly authenticate users across multiple applications, thus delivering the benefits of single sign-on without having to add additional single sign-on code
- It allows an application written in any programming language to use FIDO2 services as long as the application can call web services using the REST or SOAP protocols
- It enables web applications to take advantage of *high availability (HA)* and *disaster recovery (DR)* without having to design additional logic into the application to deliver this capability—when the StrongKey Tellaro is deployed in clusters, applications may communicate with any available node in the cluster to avail FIDO2 services
- It enables greater security since every Tellaro node has its own cryptographic hardware module to secure key management functions within the appliance (see the *Security* section of this chapter to learn more about this)

StrongKey Tellaro appliances are always installed in a clustered configuration for HA in a production environment. All the servers function in a master-master configuration where any server can receive transactions and will asynchronously replicate data records between all other nodes in the cluster without human intervention.

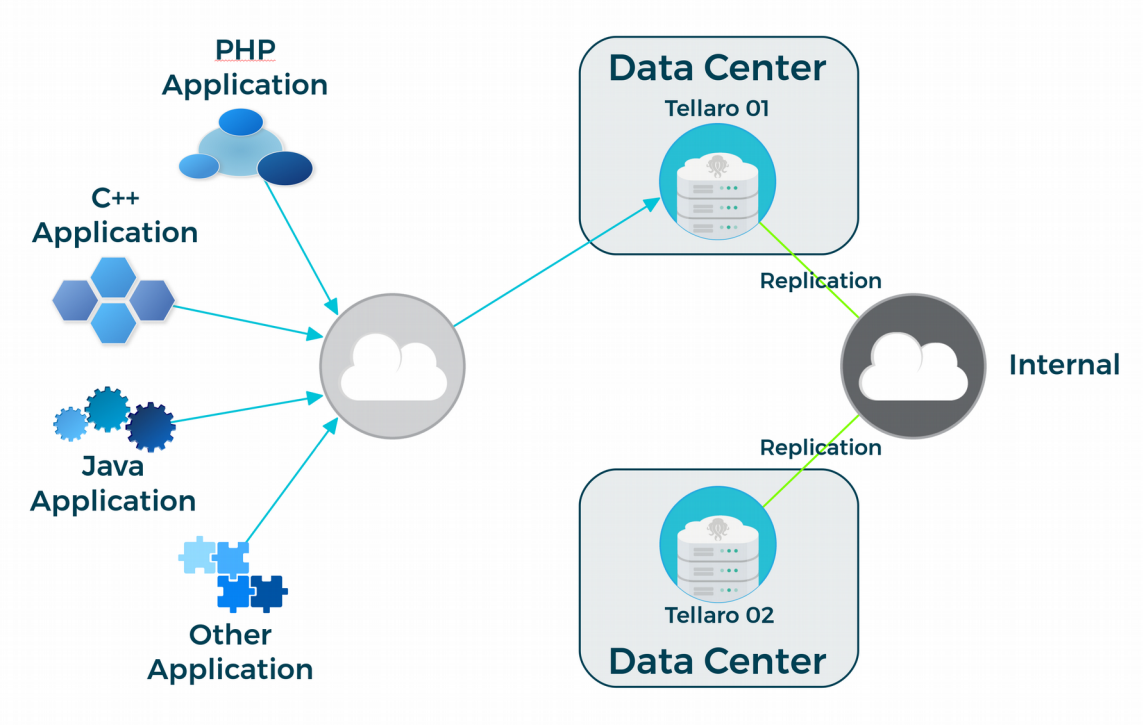
In the event a cluster node fails for any reason—network failure, power loss, hard disk crash—some "in-flight" transactions might fail, causing the registration or authentication transaction to be started over again.

While the failed node is not within the cluster, other nodes within the cluster hold all necessary transactions—such as newly registered keys—in a queue for the missing node. When the missing node rejoins the cluster, the remaining nodes in the cluster recognize its arrival and replay transactions from the replication queue to bring the missing node up to date.

When multiple Tellaro nodes are deployed across a *wide area network (WAN)* with a reasonable geographical distance between them, they also provide *business continuity (BC)* in the event of a disaster affecting a location where the services of one (set of) FIDO2 server(s) may be affected.

The diagrams in the next sections demonstrate different configurations to consider when deploying Tellaro appliances containing the FIDO2 Server.

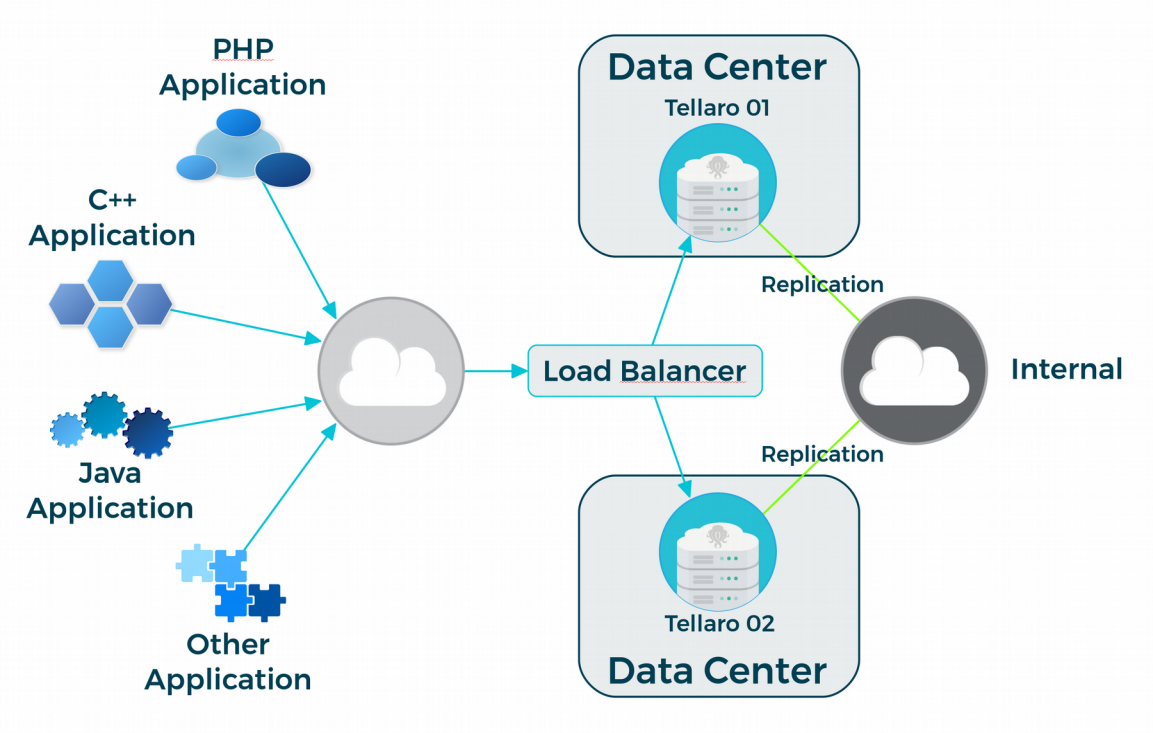
3.1.1.1—Two Nodes, Single/Multiple Data Center



Only two Tellaro appliances are deployed in a cluster, either in one data center or split between two, exposing the SOAP and REST web service API on port 8181 (the default Payara port where StrongKey web services are enabled).

In this depiction, the applications are only pointing to the primary data center (*Tellaro 01*) for FIDO2 services, which asynchronously replicates data to the other node over ports 7001, 7002, and 7003—whether it is in the same data center or over WAN. The assumption is that applications using this configuration are designed to fail over to using the second node (*Tellaro 02*) if the primary data center's node becomes unavailable for any reason. While commercial or open-source load balancers can provide this resiliency to applications without any failover code in the application, it becomes a matter of the application designer's choice on how they wish to handle application resiliency in the face of a complete failure of a cluster node or a data center.

3.1.1.2—Two Nodes, Single/Multiple Data Centers with a Load Balancer

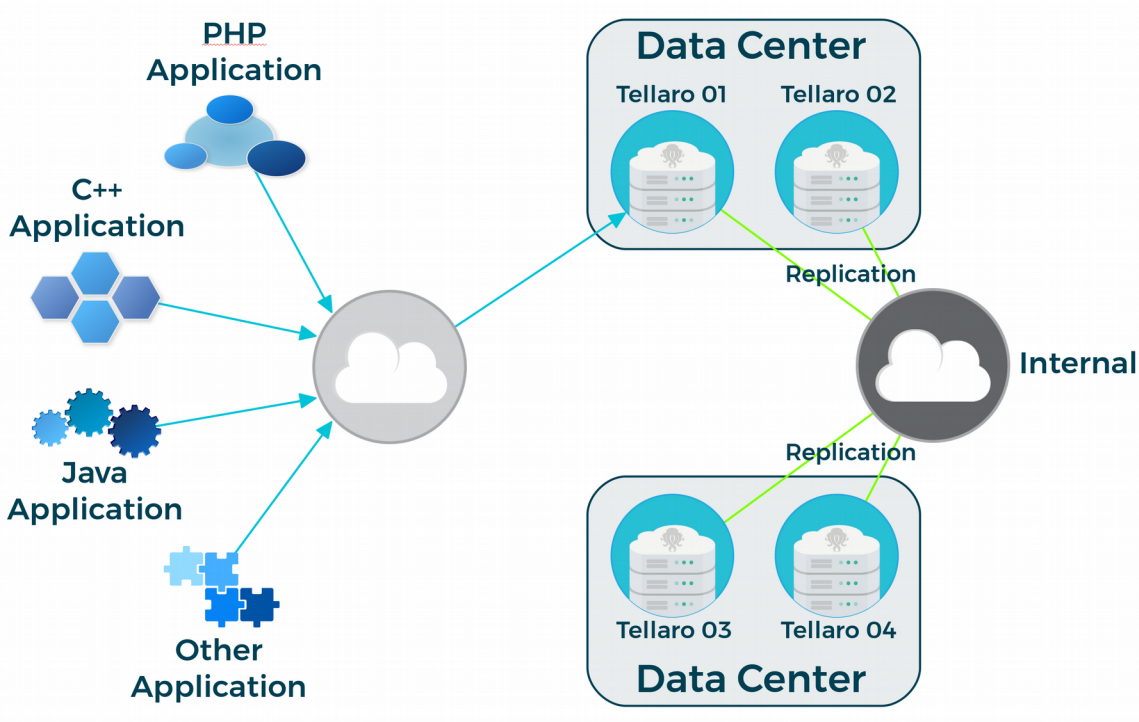


As in the first configuration, two Tellaros are deployed in one or two data centers, exposing the Tellaro's SOAP and REST web services on port 8181. The Tellaros in this scenario are behind a load balancer and the transactions can go to either Tellaro in the cluster; data is replicated asynchronously over ports 7001, 7002, and 7003.

In this configuration, application designers do not have to build in resilience to node or data center failures for their applications—the load balancer handles it for them. The applications send their web service requests to the load balancer's web service endpoint and the load balancer takes on the responsibility of managing the transaction relay to the appropriate Tellaro node based on the load balancer's rules.

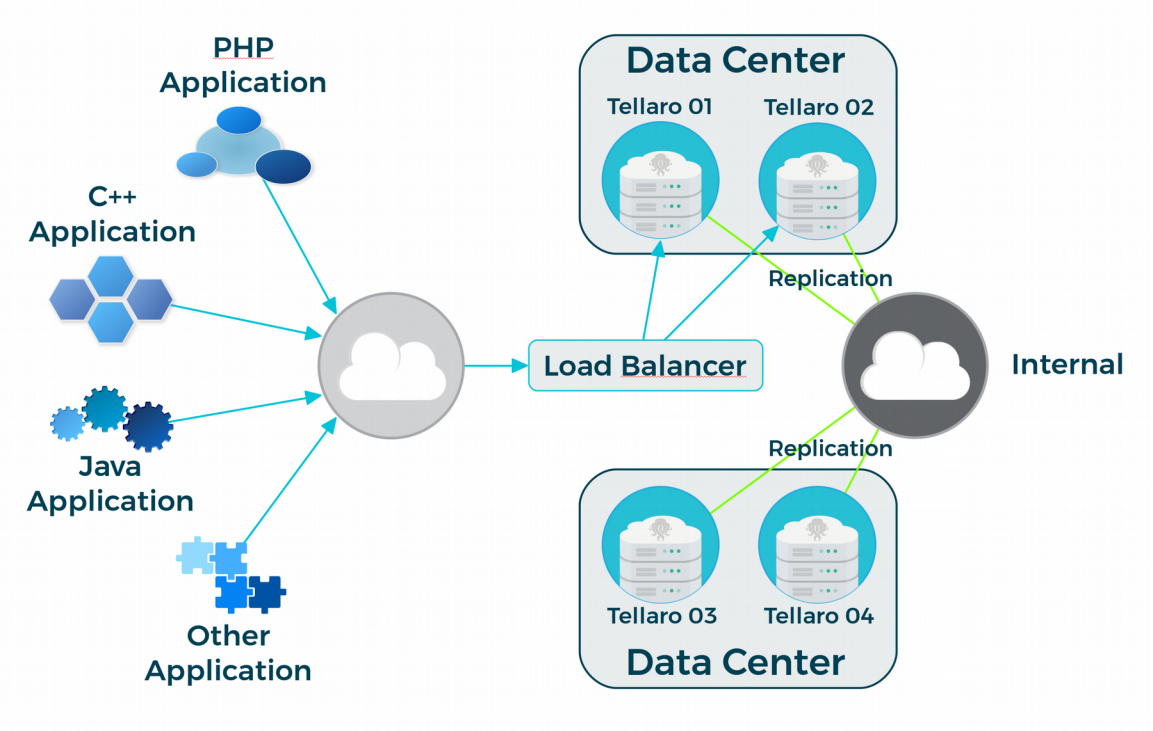
Failures of any single Tellaro are transparent to applications, since the load balancer recognizes the unavailability of the failed Tellaro and routes transactions to the available node. When the failed Tellaro rejoins the cluster, the load balancer recognizes this and starts routing transactions to the rejoined node again. The Tellaros synchronize their data automatically upon the clustered nodes seeing each other on the network again.

3.1.1.3—Four Nodes, Multiple Data Centers



This is a recommended scenario where the devices are distributed across geographical locations in two data centers and each data center also has HA of Tellaro nodes in the event of the failure of a single node in either data center. The applications are still pointing to a primary data center, replicating to the other data center asynchronously over ports 7001, 7002, and 7003.

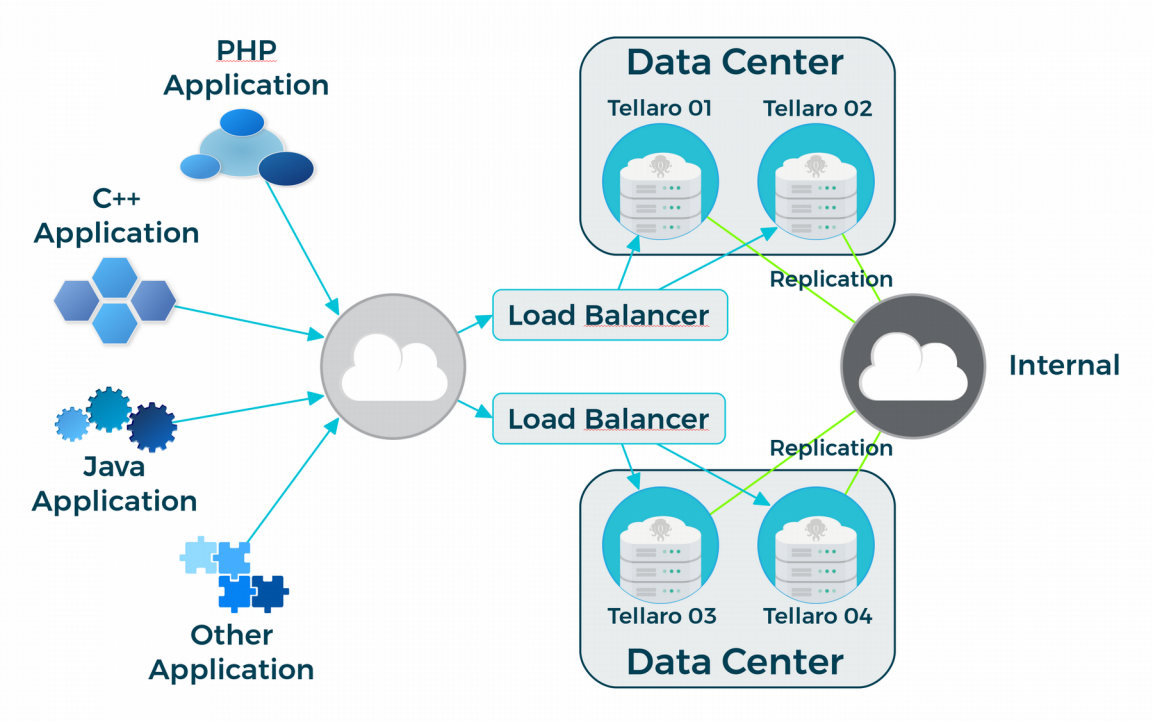
3.1.1.4—Four Nodes, Multiple Data Centers, One Load Balancer



This is another recommended scenario where the devices have been distributed across geographical locations in two data centers and each data center maintains HA for the Tellaro nodes. The devices in this scenario are behind a load balancer; the transactions can go to either device in the primary data center cluster. As before, the data are replicated asynchronously over ports 7001, 7002, and 7003.

In this configuration, the applications are shielded from having to communicate with the Tellaro nodes directly, and assume that if the entire data center (where *Tellaro 01* and *Tellaro 02* are placed) fails, the load balancer can be routed—manually or automatically—to point to the data center where *Tellaro 03* and *Tellaro 04* are located.

3.1.1.5—Four Nodes, Multiple Data Centers with One Load Balancer per Data Center



This is the highly recommended scenario for enterprise-scale organizations where the Tellaros are distributed across geographical locations in two data centers, and each data center provides HA between data centers and all Tellaro nodes. Each data center uses a load balancer for the Tellaros. Transactions may go to any device, and the data is replicated asynchronously over ports 7001, 7002, and 7003.

3.1.1.6—Considerations

- The decision with regards to which load balancer is used by the application can be made based on where the request originates, so it reaches the closest data center for a faster response
- In either of the above scenarios, it is recommended that a *Transport Layer Security (TLS)* digital certificate with a *subjectAltName (Subject Alternative Name*, or *SAN)* extension is created to include the *fully qualified domain name (FQDN)* for all Tellaros in the cluster, so that the calling applications only need to import one TLS certificate per cluster into their TrustStore
- StrongKey's implementation of the FIDO2 web services (*register* and *authenticate*) are two-step processes [see Error: Reference source not found—Error: Reference source not found] where a challenge (nonce) must be obtained in a *preregister* step before a registration, or in a *preauthenticate* step before an or authentication can be completed. If the appliances are behind a load balancer, there is a possibility that the application can go to two different machines for the two steps: *preregister* and *register* web services or the *preauthenticate* and *authenticate* web services. All challenges and sessions are stored in a map that is replicated to all other appliances in the cluster. The possibility of a small delay exists due to network problems between different data centers, or even between servers in the same data center. To compensate for such delays, developers can do one of the following:
  - Configure the load balancer to use sticky sessions so that the two related web services go to the same server for a guaranteed response
  - Design the application to handle this scenario and just retry the second *register* or *authenticate* web service again if it fails the first time
  - Build in a small delay between the two web services for replication to catch up

In an ideal scenario, the replication is instantaneous and the application does not even notice if the web services go to a different data center.

3.1.2—Configuration

The SKFS software comes with many customizable properties that do not need to be changed for most sites. However, some properties *may* be customized for sites based on their security policies. The following table presents a list of all the properties used by this release of SKFS.

Configuration properties are divided into either *Immutable* or *Mutable properties*. Immutable properties may not be changed at any time as this will impact SKFS operations. Mutable properties may be modified as needed to meet business, operational, technical, and/or security requirements.


NOTE: An Immutable property value must **never** be modified, as this will break the application, void the Support contract, and risk permanently corrupting SKFS data.

All SKFS server properties are bundled in the `resources` folder of the *fidoserverbeans* module in the SKFS application. This file provides the central configuration for the system. This central configuration file should not be modified directly. However, by modifying `skfs-configuration.properties` in the `SKFS_HOME/etc` directory, a site can override the default configuration properties bundled in the application module. An editor (gedit, nano, or vi) may be used to create/alter the configuration file in the `SKFS_HOME/etc` folder, after which Payara must be restarted for the changed properties to take effect.

3.1.2.1—Immutable Configuration

All properties use the format *key=value* where the key is of the form `skfs.cfg.property.<some-property-name>`.

Property	**skfs.cfg.property.saka.encryption.wsdlsuffix**
Explanation	The *Web Service Definition Language (WSDL)* suffix to be appended to the Tellaro URL to make a *KeyAppliance (KA)* module web service request.
Immutable Value	/strongkeyliteWAR/EncryptionService?wsdl
Property	**skfs.cfg.property.skfshome**
Explanation	The default installation directory of the SKFS software.
Immutable Value	/usr/local/strongkey/skfs
Property	**skfs.cfg.property.fido.usermetadata**
Explanation	This property determines where the FIDO users' metadata is stored. The value *local* indicates the data is stored in the Tellaro's local database in the *fido_users* table.
Immutable Value	Local
Property	**skfs.cfg.property.jdbc.jndiname**
Explanation	The *Java Naming and Directory Interface (JNDI)* name for the resource to access the MariaDB database.
Immutable Value	jdbc/strongkeylite

3.1.2.2—Mutable Configuration

Property	**skfs.cfg.property.db.keyhandle.encrypt**
Explanation	The SKFS database stores a *KeyHandle* object for every user's registered key in the *fido_keys* table. A site can choose to additionally encrypt and tokenize this value using the *Key Management (KM)* module (if available). *This property determines whether the FIDO2 Server encrypts the KeyHandle or not.*
Default Value	false
Property	**skfs.cfg.property.db.keyhandle.encrypt.saka.domainid**
Explanation	If the **skfs.cfg.property.db.keyhandle.encrypt** property is set to *true*, then this property numerically identifies the KA domain which will be used to tokenize and store the KeyHandle for every registered FIDO key.
Default Value	1

Property	skfs.cfg.property.db.signature.rowlevel.add
Explanation	To protect the integrity of records in the SKFS database, the server generates a digital signature for every row and persists the signature with the row data. This signature is verified to check row data integrity each time this row is retrieved by SKFS. This property determines whether or not all database rows have a digital signature associated with them. This feature distinguishes SKFS from other implementations. Because the FIDO protocols are designed to be “privacy protecting,” user information is not transmitted within the cryptographic messages of the protocols. The association (binding) of the registered key to a specific credential (username) within a FIDO server happens *outside* the cryptographic messages. This, unfortunately, may lead to attacks against a FIDO server implementation that go unnoticed by users and service providers (the operators of a website using FIDO protocols). An attack that uses a structured query language (SQL) injection vulnerability in a web application, or an attack that compromises the *Database Administrator (DBA)* credential—or any database credential with write access to the database schema—may insert or update critical attributes of users' registered keys. For example, by overwriting a legitimate KeyHandle with the attacker’s own KeyHandle (previously registered on that site), the attacker not only locks out everybody from the site, but allows the themselves to authenticate to anybody's account on that site with his own FIDO Authenticator. A more insidious attack is where the attacker *adds* his/her own KeyHandle as an additional registered key to every user's record in the database. This now enables the attacker to authenticate to any user's account on that site, and the legitimate user does not know her account has been compromised—unless she notices an additional “suspicious” looking registered key in their profile. SKFS generates a digital signature on every record stored in the database at the time of insertion and stores the signature with the record. The server verifies the signature each time the record is retrieved to make sure the record has not been modified since its last use. Authorized updates to the record cause the FIDO2 Server to generate a new digital signature on the modified record and store the new signature with the updated record. As a result, an attack on the database record in SKFS immediately highlights the compromise; in such a situation, besides writing warning messages in the server’s log, the server refuses to use the compromised record to authenticate the user. Applications may choose to have the user go through another authentication transaction, but this time the application may call on another FIDO2 Server in the cluster to determine if the user’s record is intact on that server. The probability of every cluster node being compromised is small, but is nonetheless possible for an insider attack. By using a signing key protected by the cryptographic hardware module on the Tellaro, StrongKey ensures that an attacker cannot successfully authenticate into another legitimate user’s account with the attacker’s own registered keys on that site. While this does reduce the number of *transactions per second (TPS)* the FIDO2 Server delivers, StrongKey believes that it is more important to be secure when attempting to use a strong authentication protocol lest companies be lulled into a false sense of security.
Default Value	true

Property	skfs.cfg.property.db.signature.rowlevel.verify
Explanation	This property determines whether or not the server will verify row-level signatures. It should only be set to *true* if the **skfs.cfg.property.db.signature.rowlevel.add** property is set to *true*. If the ***signature.*.add** property is set to *false* and the ***signature.*.verify** property is set to *true*, all authentication transactions will fail since the server will be unable to find a digital signature for the user in her record.
Default Value	true
Property	skfs.cfg.property.entropylength
Explanation	FIDO2 Server is responsible to generate a challenge (nonce) for all registration and authentication requests. This property determines the length of entropy to be used for generating these challenges.
Default Value	512
Property	skfs.cfg.property.fido2.user.settings.version
Explanation	This property determines the version settings for FIDO2 key registration. This property is present for future-proofing the code. **It is not recommended that this value be changed in this version.**
Default Value	1
Property	skfs.cfg.property.fidokeys.flush.cutofftime.seconds
Explanation	To speed up FIDO2 transaction processing, the server temporarily caches registered keys in memory. This property determines the maximum number of seconds a key can remain cached in memory. The longer a key is cached in memory, the more memory is required within the Tellaro. Please keep in mind that once a user has authenticated to the FIDO2 Server, they are not likely to need the key again for awhile. The default value is useful when a user registers a new key with the server and immediately attempts to authenticate with that key. In that situation, this property is useful to speed up the authentication transaction.
Default Value	30
Property	skfs.cfg.property.fidokeys.flush.frequency.seconds
Explanation	To speed up FIDO2 transaction processing, the server temporarily caches registered keys in memory. This property determines the frequency at which a server thread responsible for flushing keys out of memory wakes up to perform its work.
Default Value	5
Property	skfs.cfg.property.fido.userid.length
Explanation	Determines the maximum allowed length for a credential's username for a user account. When changing it, **never** reduce the size once users have started registering themselves. Either reduce it before users start registering, or consider using a different cryptographic domain for different applications that have different needs.
Default Value	32

Property	skfs.cfg.property.messaging.blpsleeptime
Explanation	Before an object is replicated to other appliances, the source appliance saves the object metadata in the *replication* table, and then publishes the object to subscribers. Sometimes, the acknowledgment from subscribers may not reach the publisher. On such occasions, the publisher has a *BacklogProcessor (BLP)* that attempts to resend the object to subscribers who did not receive it. However, to ensure that the BLP does not get caught in an endless loop sending the objects continuously, it sleeps for the number of seconds specified in this parameter before checking the *replication* table again to publish objects.
Default Value	60
Property	skfs.cfg.property.messaging.timediff
Explanation	ZeroMQ normally replicates most objects to all appliances when they are created. However, there are occasions when some leftover objects might remain in the replication table that were either not acknowledged by recipients, or the publisher did not receive the acknowledgment as it was too busy. In these situations, the BLP attempts to replicate the object again as part of the clean up processor. The *timediff* property is the amount of seconds a record must be in the *replication* table before the BLP attempts to resend it to other appliances.
Default Value	60
Property	skfs.cfg.property.pkix.validate.default.truststore.password
Explanation	[Not yet implemented] Indicates the default password used by the server for its TrustStore containing the root certificates for validating the attestations provided by FIDO registrations.
Default Value	changeit
Property	skfs.cfg.property.pkix.validate.default.truststore
Explanation	[Not yet implemented] Indicates the default TrustStore used for *Public Key Infrastructure (X.509) (PKIX)* validation.
Default Value	/usr/local/strongkey/skfs/etc/pkix-truststore.jceks
Property	skfs.cfg.property.pkix.validate.method
Explanation	[Not yet implemented] The default mechanism of validating attestation certificates.
Default Value	TrustStore
Property	skfs.cfg.property.pkix.validate
Explanation	[Not yet implemented] This property indicates whether or not attestation certificates must be validated.
Default Value	true
Property	skfs.cfg.property.retrieve.tld
Explanation	FIDO U2F protocol has a concept of APPID-FACETID verification which involves checking *Top Level Domains (TLD)*. This property indicates whether the TLD list should be fetched from an external URL. *This property can only be true if the server can connect to the internet.*
Default Value	false

Property	skfs.cfg.property.standalone.fidoengine
Explanation	This property determines if the FIDO2 Server has been deployed in a standalone setup or as a part of the Enterprise KA deployment.
Default Value	true (for the GitHub release) false (for the Enterprise Server)
Property	skfs.cfg.property.standalone.signingkeystore.password
Explanation	If skfs.cfg.property.standalone.fidoengine has been set to *true*, the server may not have access to a hardware so it uses a keystore for storing database row level signing keys. This property indicates the default password used for this keystore.
Default Value	Abcd1234!
Property	skfs.cfg.property.usersession.flush.cutofftime.seconds
Explanation	To speed up FIDO transaction processing, the server caches user session information in memory temporarily. This property determines the maximum time any key can remain in memory.
Default Value	30
Property	skfs.cfg.property.usersession.flush.frequency.seconds
Explanation	To speed up FIDO transaction processing, the server caches user session information in memory temporarily. This property determines the frequency of execution for the FIDO keys clean up job.
Default Value	5

3.1.3—Changing Configuration Properties

The following three commands are for domains-specific changes only. Global changes are covered in the next section.

3.1.3.1—Get Configuration

Get all current admin configurations for appliance, LDAP, and SKFS. Explains what each configuration is used for.

Example:

```
java -jar skfsclient.jar GC https://example.strongkey.com:8181 1 REST PASSWORD
fidoadminuser Abcd1234!
```

Example Response (*GetConfiguration*):

```
GetConfiguration response :
{
    "appliance": [
        {
            "configkey": "appl.cfg.property.service.ce.ldap.ldaptype",
            "configvalue": "LDAP",
            "hint": "Property that identifies what type of LDAP will be used
for authenticating service credentials for the domain. Acceptable values :
LDAP | AD. Default value: LDAP  "
        }
    ],
```

```
    "ldap": [
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapadmingroup",
            "configvalue": "cn=AdminAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
Administrator group in LDAP/AD. Default value : cn=AdminAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapcloudmovegroup",
            "configvalue": "cn=CloudMoveAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
file move authorized group in LDAP/AD. This property is only used by the
file encryption module. Default value : cn=CloudMoveAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapdecryptiongroup",
            "configvalue": "cn=DecryptionAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
file decryption authorized group in LDAP/AD. This property is only used by
the file encryption module. Default value : cn=DecryptionAuthorized"
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldapdnprefix",
            "configvalue": "cn=",
            "hint": "Property that identifies the Distinguished name (DN)
prefix to be used for service credentials. Default value : cn="
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldapdnsuffix",
            "configvalue":
",ou=users,ou=v2,ou=SKCE,ou=StrongAuth,ou=Applications,dc=strongauth,dc=com"
,
            "hint": "Property that identifies the user suffix to be appended
to the user dn. Default value :
,ou=users,ou=v2,ou=SKCE,ou=StrongAuth,ou=Applications,dc=strongauth,dc=com"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapencryptiongroup",
            "configvalue": "cn=EncryptionAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
file encryption authorized group in LDAP/AD. This property is only used by
the file encryption module. Default value : cn=EncryptionAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapfidoadmingroup",
            "configvalue": "cn=FidoAdminAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
FIDO admin authorized group in LDAP/AD. This property is only used by the
```

```
FIDO server to perform admin (policy and configurations) operations. Default
value : cn=FidoAdminAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapfidoauthzgroup",
            "configvalue": "cn=FidoAuthzAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
FIDO authorizations authorized group in LDAP/AD. This property is only used
by the FIDO server to perform pre-authorize and authorize operations.
Default value : cn=FidoAuthzAuthorized"
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldapfidogroup",
            "configvalue": "cn=FidoAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
FIDO authorized group in LDAP/AD. This property is only used by the FIDO
server to perform patch and delete operations. Default value :
cn=FidoAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapfidoreggroup",
            "configvalue": "cn=FidoRegAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
FIDO registration authorized group in LDAP/AD. This property is only used by
the FIDO server to perform pre-register and register operations. Default
value : cn=FidoRegAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapfidosigngroup",
            "configvalue": "cn=FidoSignAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
FIDO assertion authorized group in LDAP/AD. This property is only used by
the FIDO server to perform pre-authenticate and authenticate operations.
Default value : cn=FidoSignAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapgroupsuffix",
            "configvalue":
",ou=groups,ou=v2,ou=SKCE,ou=StrongAuth,ou=Applications,dc=strongauth,dc=com
",
            "hint": "Property that identifies the groups suffix to be
appended to the group dn. Default value :
,ou=groups,ou=v2,ou=SKCE,ou=StrongAuth,ou=Applications,dc=strongauth,dc=com"
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldaploadgroup",
            "configvalue": "cn=LoadAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
Key Load authorized group in LDAP/AD. This property is only used by the
```

```
signing module. Default value : cn=LoadAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapremovegroup",
            "configvalue": "cn=RemoveAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
Key remove authorized group in LDAP/AD. This property is only used by the
signing module. Default value : cn=RemoveAuthorized"
        },
        {
            "configkey":
"ldape.cfg.property.service.ce.ldap.ldapservicegroup",
            "configvalue": "cn=Exampleeeee",
            "hint": "Property that identifies the Common Name (CN) for the
Services group in LDAP/AD. Default value : cn=Services"
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldapsigngroup",
            "configvalue": "cn=SignAuthorized",
            "hint": "Property that identifies the Common Name (CN) for the
Sign authorized group in LDAP/AD. This property is only used by the signing
module. Default value : cn=SignAuthorized"
        },
        {
            "configkey": "ldape.cfg.property.service.ce.ldap.ldapurl",
            "configvalue": "ldap://localhost:1389",
            "hint": "Property that identifies the LDAP/AD url for the
authentication/athorization of service credentials. DEfault value :
ldap://localhost:1389"
        }
    ],
    "skfs": [
        {
            "configkey": "skfs.cfg.property.fido2.user.sendfakeKH",
            "configvalue": "false",
            "hint": "Property that identifies if fake keyhandles should be
sent back to the calling application when they request preauthentication for
unregistered users. Accepted Values : TRUE | FALSE. Default value : FALSE"
        }
    ]
}
```

3.1.3.2—Update Configuration

Update any of the keys shown in the above JSON with the new specified value.

Example:

```
java -jar skfsclient.jar UC https://example.strongkey.com:8181 1 REST PASSWORD
fidoadminuser Abcd1234! ldape.cfg.property.service.ce.ldap.ldapservicegroup
cn=Example
```

3.1.3.3—Delete Configuration

Deletes any of the keys shown in the above JSON.

Example:

```
java -jar skfsclient.jar DC https://example.strongkey.com:8181 1 REST PASSWORD
fidoadminuser Abcd1234! ldape.cfg.property.service.ce.ldap.ldapservicegroup
```

3.1.4—Global Configuration

To make global configuration changes, modify configurations in the appropriate location of for the application in question:

- StrongKey FIDO2 Server (SKFS):
  `/usr/local/strongkey/skfs/etc/skfs-configuration.properties`
- Tellaro KeyAppliance (KA):
  `/usr/local/strongkey/appliance/etc/appliance-configuration.properties`
- Tellaro CryptoEngine (CE):
  `/usr/local/strongkey/skce/etc/skce-configuration.properties`

3.2—Security

This chapter provides information about how SKFS is secured and what a site needs to be aware of to preserve the security of that environment. All commands in this chapter have been implemented in a CentOS environment; please take this into account if using a different operating system than CentOS.

3.2.1—Operating System

Since SKFS uses the Linux operating system, all security settings indicated below pertain to the Linux operating system. A site may supplement the standard Linux capabilities with third-party security tools, if their security policy requires it.

Sites are responsible for ensuring that the addition of third-party tools to SKFS do not violate the security settings created during system setup. Any reduction in SKFS operating system security can create potential vulnerabilities which void StrongKey's warranty on the appliance.

The Linux firewall, *iptables*, is configured to only make port 22 (for *Secure Shell (SSH)*) and port 8181 (for the SKFS *EncryptionService*) accessible over the network using the *Transport Layer Security (TLS)* protocol. System Administrators require the use of port 22 for administering the machine remotely, while applications will access the SKFS web service over port 8181. Additionally, ports 7001, 7002, and 7003 are opened selectively between SKFS nodes to enable database replication. Clients outside the SKFS cluster will not be able to access these ports. All other ports, including the *Internet Control Message Protocol (ICMP)*, are blocked from the machine. The denial of ICMP traffic implies that the SKFS server cannot even be pinged on the network. Network Administrators will need to determine if port 8181 is accessible to check for network connectivity to the appliance.

The SKFS installation process creates some application accounts, but locks all accounts not necessary for the administration of the appliance. This minimizes user access to the Linux operating system from the console or remotely over SSH to just two administrative accounts—*root* and *strongkey*.

Please consult Linux documentation for how to enhance the security of your appliance above and beyond what is provided in the base installation.

3.2.1.1—Network

Since the only two ports that are publicly visible on the network are port 22 (SSH) and 8181 (SKFS *EncryptionService),* all data traversing the network to these two ports are protected from attacks on the wire. SSH encrypts data with a randomly generated session key, while SKFS uses *Transport Layer Security (TLS)* 1.3, which also protects data with a randomly generated session key. Both applications use public-key cryptography to protect the session keys on the network.

The SKFS SSH key pair is generated during the installation of the Linux operating system, and its keys are protected using operating system controls. The TLS key pair and digital certificate for the SKFS service are generated during the installation of the *Java Enterprise Edition (JEE)* application server and are protected using operating system controls and a password.

3.2.1.2—Relational Database

SKFS uses a standard *relational database management system (RDBMS)* to store data about encrypted objects and decryption requests.

The database service is accessible only from the local machine. Remote clients and applications will be unable to access the database directly because of the firewall controls on the operating system. This does not prevent the SKFS application from accessing the database locally.

While there are no special requirements for database security (other than what may be required of a site's security policy for Production databases), it is recommended that backups of the database are stored separately from standard backups in the event that vulnerabilities in the encryption algorithm are discovered in the future. Controlled access to database backups will minimize any damage from such potential discoveries.

3.2.1.3—JEE Application Server

SKFS uses a standard JEE application server to process web service requests. While the application server hosts the application, nothing in the configuration of the application server allows someone to gain access to sensitive data in the database.

3.2.1.4—Lightweight Directory Access Protocol (LDAP)

SKFS is capable of integrating with LDAP on the network to authenticate and determine the authorization of users who request its web services. The LDAP service may either be an industry standard LDAP service or Microsoft Active Directory.

When a user requests a register or authenticate web service from SKFS, they must pass to the web service an LDAP username and password with the data. The *SKFSServlet* uses this username and password to authenticate to the LDAP service and then determines whether the user is a member of one of two LDAP groups—the *FIDORegAuthorized* or *FIDOSignAuthorized* groups—based on the type of service requested by the user. Only when the two LDAP checks pass does SKFS continue with the performance of the service.

If the LDAP service is not protected on the network using either TLS or *Internet Protocol Security (IPSec)*, there is a possibility that attackers may snoop the LDAP credentials between the SKFS server and the LDAP server. The compromise of these credentials will allow attackers to legitimately request cryptographic web services from the SKFS server using the compromised credentials.

It is strongly recommended that sites either use the LDAP over TLS capability, or tunnel the LDAP service over IPSec. In either case, the encryption on the wire will protect the user credentials being sent to the LDAP service for authentication.

3.2.2—Appliance Credentials

This section describes various accounts used in SKFS.

The most important element of cryptographic security is to protect cryptographic keys from unauthorized entities. All unauthorized entities, when launching an attack on a system, must compromise some credential of the system before attempting to gain access to cryptographic keys. No matter which credential gets compromised first, the target credential required by attackers is the *root* superuser of the system or the *strongkey* owner of the application—the credential authorized to execute the web service application's cryptographic functions. The web service *applicationID* has direct access to key material, while the superuser—with the ability to assume any user's identity on a system, including that of the *applicationID*—has indirect access.

Consequently, protecting data on the SKFS server comes down to ensuring that the web service application credential is protected, while placing strong controls on the superuser of the system.

3.2.2.1—Credential Matrix

The following table describes the credentials used on SKFS and their access to the keys on the hardware module. It helps in summarizing the degree of vulnerability each SKFS credential can have on the keys and sensitive data in the database.

Credential	Purpose	Access to Keys?
Linux *root*	The superuser of the operating system normally has unlimited control over everything on the computer system, but without the required PINs for authentication to the hardware module, even the *root* user is incapable of accessing keys on the module. Because of the privileged status of the *root* user, he/she can assume the identity of any user on the Linux operating system. As a result, additional controls must be placed around this credential if the site security policy wishes to restrict individuals (with access to *root*) from accessing cryptographic keys on the hardware module. While the controls described in the *Protecting PINs from root* section of this document provides one method of restricting this access, sites must ultimately use methods that are in compliance to their internal security policies.	Indirectly
Linux *strongkey*	This operating system user ID owns the web service application software that comprises SKFS. Since the web service application uses this user ID within the Linux operating system, this account has access to the PINs for the hardware module (so it can authenticate to the module before performing cryptographic operations on data). Care must be taken to secure this account and only provide its credential to trusted administrators.	Yes

Credential	Purpose	Access to Keys?
MariaDB *root*	This superuser account of the SKFS RDBMS has full control over everything inside all databases on the system. However, since data stored in the SKFS database only consists of ciphertext (data encrypted by the SKFS web application before it reaches the database) without any sensitive material that could decrypt them, this user cannot decrypt ciphertext despite their full control over database data.	No
MariaDB *skfsdbuser*	The owner of database schemas and data in the SKFS internal database has unlimited control over everything inside this database schema and objects. However, as already indicated, since data stored in this internal database only consists of ciphertext without any sensitive material that could lead to its decryption, this user cannot decrypt ciphertext.	No
Payara *admin*	This administration account of the JEE application server has full control over the application server's configuration. However, since this is not an operating system account, this administrator cannot read or modify anything other than the application server configuration files. This credential only exists to facilitate the administration of the application server through a web interface.	No
Web Service Credential	This credential is used by SKFS to authenticate a requester of web services. Depending on the SKFS configuration, this can be against the SKFS managed users or against an LDAP-based Directory Server. This credential is verified by the SKFS web service application before it performs the cryptographic operation. Since the credential has no operating system, RDBMS, application server, or hardware module privileges, this user cannot access cryptographic keys in the module. If this credential has the *decryption* privilege, it will have the ability to request decryption services from SKFS. As a consequence, the password to this credential must be protected by the applications that will include them in their web service request to SKFS; SKFS does not maintain this credential's password.	Indirectly

3.2.2.2—Protecting against *root*

The *root* credential is the most powerful credential on a Linux/UNIX computer. If this credential is compromised, it is generally considered a full system compromise. It is generally recommended to have user applications run WITHOUT *root* privileges—if the application has bugs or is exploited, it will not give up privileged access.

SKFS practices this defense strategy by not requiring or using *root* privileges for normal operations—some parts of the installation process require *root* privileges, but once completed, the web service application that provides cryptographic services does not need *root* access. Because any human (or attacker) that has access to the *root* account can assume the identity of any other Linux account, it would be possible for an attacker who has compromised the *root* account to assume the SKFS web service application ID.

As a consequence, sites may want to ensure that the *root* account is restricted through policy, procedural, and technical controls. While this document does not go into the policy and procedural aspects of controlling *root* access, the following technical controls can help mitigate some of the risks of the *root* account becoming compromised. Site administrators are encouraged to review other resources to determine the optimum mechanism for protecting and/or controlling *root* privileges on their systems, so they gain a broader perspective on these controls.

1. Setup an *unlock* account on the Linux system. This account becomes the fail-safe measure to recover access to the *root* account in the event it is needed for extraordinary administrative actions.
2. Add the *unlock* account to `/etc/sudoers` with the ability to unlock the *root* account. Replace the `<FQDN>` with the fully qualified domain name of the SKFS server.

   ```
   unlock <FQDN>=/usr/sbin/usermod --unlock root
   ```

3. Add the normal Linux account of the *Systems Administrator (SA)* to `/etc/sudoers` to perform specific SA tasks (as defined by company policy). This ensures that SAs can perform their day-to-day jobs—backup, restore, log management, performance management, patching, etc. These accounts must be the only legitimate means of managing the SKFS system.

   For tighter control, depending on company policy, sites may add multi-factor authentication tokens to the system to ensure only legitimate SAs with two-factor tokens can login into SA accounts in SKFS.

4. Add a watch on `/etc/sudoers` and on `/usr/bin/sudo` to audit modifications to the former and use of the latter:

```
auditctl -w /etc/sudoers -pwar -k sudoers
auditctl -w /usr/bin/sudo -px -k sudo-exec
```

   This first watch monitors for writes, appends, and reads of the `/etc/sudoers` file, then logs them. Logged records can be searched with filter key `sudoers`. The second watches for execution of the `sudo` command and log any matches with a filter key of `sudo-exec`.

   `auditd` should be turned on to monitor for access to the `sudo` command and report it to appropriate authorities for review (third-party reporting tools may also be used for this purpose, if desired).

5. Finally, lock the *root* account on the system:

```
/usr/sbin/usermod --lock root
```

   This has the effect of disabling the *root* account so no one can gain access to the account.

3.2.2.3—Protecting the *strongkey* Application Credential

The *strongkey* credential is the only one that has direct access to cryptographic keys. This access is enabled through the web service application that provides FIDO services to requesting applications.

While the *strongkey* credential does not require any other privileged access on the system (that may compromise the system), it must be protected as diligently as the privileged *root* account of the system.

The following controls can help protect the *strongkey* account:

Setup an *unlock* account on the Linux system (if it hasn't already been created for controlling the *root* account).	This account becomes the fail-safe measure to recover access to the *strongkey* account in the event it is needed for extraordinary administrative actions.
Add the *unlock* account to `/etc/sudoers` with the ability to unlock the *root* account.	Replace the `<FQDN>` with the fully qualified domain name of the SKFS server: `unlock <FQDN>=/usr/sbin/usermod --unlock strongkey`
Add the normal Linux account of the Systems Administrator to `/etc/sudoers` to perform SKFS-specific SA tasks (as defined by company policy).	This ensures that SAs can perform their day-to-day jobs with the SKFS software—backup, restore, log management, performance management, patching, etc. These accounts must be the only legitimate means of managing the SKFS software. For tighter control, depending on company policy, sites may add multi-factor authentication tokens to the system to ensure only legitimate SAs with two-factor tokens can login into SA accounts on SKFS.

Add these watches to audit modifications to various configuration files.	This first watch listed below will monitor for writes, appends, and reads of the `/etc/sudoers` file and log them. Logged records can be searched with the filter key `sudoers`. The second will watch for executions of the `sudo` command and log them with a filter key of `sudo-exec`. `auditd` should be turned on to monitor access to the `sudo` command for appropriate authorities for review (third-party reporting tools may also be used for this purpose, if desired). The generic `auditctl` command is: `auditctl -w <file> -pwar -k <filter key>`

<file>	<filter key>
skfs_HOME/etc/skfs-configuration.properties	skfs-configuration
skce_HOME/etc/skce-configuration.properties	skce-configuration
appliance_HOME/etc/appliance-configuration.properties	appliance-configuration
GLASSFISH_HOME/domains/domain1/config/domain.xml	glassfish-config

Finally, lock the *strongkey* account on the system:	This has the effect of disabling the *strongkey* account so no one can gain access to the account. `/usr/sbin/usermod --lock strongkey`

3.2.2.4—Other Controls

In addition to the above-mentioned controls, sites should turn on System Accounting to track every command executed by logged-in users (if any). While this feature will not prevent someone from compromising the system, it will provide forensic data on how the system was compromised and through which account. Third-party controls may also be used by sites to track such activity.

3.2.3—Monitoring SKFS

To make sure SKFS is running as expected, we highly recommend to regularly monitor the server using the *ping* web service. Use the sample client application `skfsclient.jar` (installed as part of the FIDO2 server installation) to call the *ping* web service.

NOTE: On the FIDO2 server, the client application can be found under the `/usr/local/strongkey/skfsclient` directory.

Different methods are listed here:

- Ping SKFS using REST and HMAC authorization:

```
shell> java -jar skfsclient.jar P https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4
```

- Ping SKFS using REST and PASSWORD authorization:

```
shell> java -jar skfsclient.jar P https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234!
```

- Ping SKFS using REST and PASSWORD authorization:

```
shell> java -jar skfsclient.jar P https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234!
```

- Ping SKFS using SOAP and PASSWORD authorization:

```
shell> java -jar skfsclient.jar P https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234!
```

Successful responses will be similar to the image above.

3.3—Adding Access/Secret Keys

This section applies only when using HMAC authentication for a FIDO server that is not part of the Tellaro appliance suite.

3.3.1—Creating a New Access Key and Secret Key

The *StrongKey FIDO2 Server (SKFS)* contains a list of pairs of access and secret keys stored in a keystore file. In order to perform FIDO2 operations, the correct access and secret keys must be used by the servlet to talk to the FIDO2 Server.

In this example we will be using the FIDO Server and PoC found at these URLs:

- https://github.com/strongkey/fido2
- https://github.com/StrongKey/fido2/tree/master/sampleapps/java/poc/

With the FIDO2 Server and POC are both installed correctly, we may begin the process of creating new a new pair of access key and secret key.

3.3.2—Adding Access/Secret Keys in Standalone SKFS

1. On the machine where the FIDO2 Server was installed, **change directory** to where `fido2server-v#.#.#` was extracted (*Step 4* of the *FIDO2 Server Installation Guide*). keymanager lives in `/usr/local/strongkey/keymanager` directory.
2. **Change directory** to the to the `keymanager` directory. Inside is the `keymanager.jar` file which will help in showing, adding, and deleting access keys. For a list of possible operations, **run the following command**:

   ```
   > java -jar keymanager.jar
   ```

3. **Create an access key** by running the command:

   ```
   > java -jar keymanager.jar addaccesskey <keystore locations>
     <keystore password>
   ```

   Assuming the keystore location is the in the same place after installation, your command will look like this:

   ```
   > java -jar keymanager.jar addaccesskey
     /usr/local/strongkey/skfs/keystores/signingkeystore.bcfks Abcd1234!
   ```

   You will get a response similar to this:

   ```
   > Created new access/secret key:
   > Access key:f38158b564b57fee
   > Secret key:4ca0275d5f7217ce5044352f40752558
   ```

   Temporarily **store the generated access key and secret key**.

4. On the machine where the PoC servlet was installed, **change directory** to `/usr/local/strongkey/poc/etc/` and use any text editor to **edit** the `.properties` file contained in the directory. **Add the following lines** to the file:

   ```
   > poc.cfg.property.accesskey=<access key>
   > poc.cfg.property.secretkey=<secret key>
   ```

It should look similar to this:

```
> poc.cfg.property.accesskey=f38158b564b57fee
> poc.cfg.property.secretkey=4ca0275d5f7217ce5044352f40752558
```

Save your changes.

5. After saving these changes, Payara must be restarted. Run the following command:

```
> sudo service glassfishd restart
```

You have now added a new access/secret key pair to your FIDO2 Server.

3.3.3—Adding Access/Secret Keys in an SKFS Cluster:

1. In one of the FIDO Servers, **complete the steps mentioned above**.
2. When finished, **copy and replace** the keystore file `signingkeystore.bcfks` from the FIDO2 Server where access/secret keys were created onto the other FIDO2 Servers in the cluster.

3.4—Test FIDO2 V3 API

This document provides information about the sample client application called *skfsclient* which is used to test StrongKey's FIDO2 Server running on the v3 API. The sample client is a *command line interface (CLI)* based client written in Java programming language; tested on *Java Development Kit (JDK)* 8.

Use *skfsclient* to test registering a key, authenticating a key, getting keys tied to a user, updating a key, and de-registering (deleting) a key.

NOTE: A full JDK installation (JDK 8 or above) is needed and just the *Java Runtime Environment (JRE)* is not sufficient.

3.4.1—Usage

1. Open a terminal window.
2. Change directory to where skfsclient is present. The skfsclient file is named `skfsclient.jar`.

   ```
   > cd /usr/local/strongkey/skfsclient
   ```

3. Execute the sample client to see the usage. Type the command below to see the usage of the tool:

   ```
   java -jar /usr/local/strongkey/skfsclient/skfsclient.jar
   ```

   Here you are given a list of operations as well as a brief description of each argument. Below is a table that shows this information.

Commands	R, A, G, U, D, P
Registration (R)	`java -jar skfsclient.jar R <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ] <username> <origin>`
Authentication (A)	`java -jar skfsclient.jar A <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ] <username> <origin> <authcounter>`
Getkeysinfo (G)	`java -jar skfsclient.jar G <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ] <username>`
Update (U)	`java -jar skfsclient.jar U <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ] <random-id> <displayname> <Active/Inactive>`
De-register (D)	`java -jar skfsclient.jar D <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ] <random-id>`

Commands	R, A, G, U, D, P
Ping (P)	`java -jar skfsclient.jar P <hostport> <did> <wsprotocol> <authtype> [ <accesskey> <secretkey> | <svcusername> <svcpassword> ]`
hostport	Host and port to access the FIDO SOAP and REST formats: `http://<FQDN>:<non-ssl-portnumber>` or `https://<FQDN>:<ssl-portnumber>` Example: `https://fidodemo.strongauth.com:8181`
wsprotocol	Web socket protocol; example REST | SOAP
authtype	Authorization type; example HMAC | PASSWORD
accesskey	Access key for use in identifying a secret key.
secretkey	Secret key for HMACing a request.
svcusername	Username used for PASSWORD-based authorization.
svcpassword	Password used for PASSWORD-based authorization.
username	Username for registration, authentication, or getting keys info.
origin	Origin to be used by the FIDO client simulator.
authcounter	Authorization counter to be used by the FIDO client simulator
random-id	String associated to a specific FIDO key registered to a specific user. This is needed to perform actions on the key, like deactivate, activate and de-register. RandomIDs can be obtained by using the *G* option.
Active/Inactive	Status to which to set the FIDO key.

The current defaults for HMAC- and PASSWORD-based authentication are as follows:

HMAC
accesskey = 162a5684336fa6e7
secretkey = 7edd81de1baab6ebcc76ebe3e38f41f4

PASSWORD
svcusername = svcfidouser
svcpassword = Abcd1234!

3.4.2—Perform Example Tests (REST and HMAC)

1. **Register a new FIDO key** to a user named *johndoe* using REST and HMAC authorization.

```
java -jar skfsclient.jar R https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://
[FQDN]:8181
```

2. **Authenticate** the same user with the FIDO key registered in the step above using REST and HMAC authorization. Provide an authentication counter starting at 1, since this is the first authentication.

```
java -jar skfsclient.jar A https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://
[FQDN]:8181 1
```

3. **Retrieve the information** about the list of FIDO keys registered by a specific user using REST and HMAC authorization.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe
```

NOTE: Each registered key can be deactivated (FIDO key would be temporarily unusable), reactivated, renamed, and de-registered (permanently deleted). Each of these operations on a specific user-registered FIDO key can be acted upon using the *random-id* value that is sent back when the key's information is retrieved from the server's response in step above (step 3).

4. **Update key information** for a specific key using REST and HMAC authorization. In this call you will be able to set the display name and deactivate or reactivate a specific FIDO key.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 1-1-testuser-171
new-display-name Active
```

5. **De-register** (delete) a specific key using REST and HMAC authorization.

```
java -jar skfsclient.jar D https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 1-1-testuser-171
```

6. **Ping** the FIDO2 Server using REST and HMAC authorization.

```
java -jar skfsclient.jar P https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4
```

3.4.3—Perform Example Tests (REST and PASSWORD)

1. **Register a new FIDO key** to a user named *johndoe* using REST and PASSWORD authorization.

```
java -jar skfsclient.jar R https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe https://[FQDN]:8181
```

2. **Authenticate** the same user with the FIDO key registered in the step above using REST and PASSWORD authorization. Provide an authentication counter starting at 1, since this is the first authentication.

```
java -jar skfsclient.jar A https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe https://[FQDN]:8181 1
```

3. **Retrieve the information** about the list of FIDO keys registered by a specific user using REST and PASSWORD authorization.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe
```

4. **Update key information** for a specific key using REST and PASSWORD authorization. In this call you will be able to set the display name and deactivate or reactivate a specific FIDO key.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! 1-1-testuser-171 new-display-name Active
```

5. **De-register** (delete) a specific key using REST and PASSWORD authorization.

```
java -jar skfsclient.jar D https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! 1-1-testuser-171
```

6. **Ping** the FIDO2 Server using REST and PASSWORD authorization.

```
java -jar skfsclient.jar P https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234!
```

3.4.4—Perform Example Tests (SOAP and HMAC)

1. **Register a new FIDO key** to a user named *johndoe* using SOAP and HMAC authorization.

```
java -jar skfsclient.jar R https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://
[FQDN]:8181
```

2. **Authenticate** the same user with the FIDO key registered in the step above using SOAP and HMAC authorization. Provide an authentication counter starting at 1, since this is the first authentication.

```
java -jar skfsclient.jar A https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://
[FQDN]:8181 1
```

3. **Retrieve the list of FIDO keys** registered by a specific user using SOAP and HMAC authorization.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe
```

4. **Update key information** for a specific key using SOAP and HMAC authorization. In this call you will be able to set the display name and deactivate or reactivate a specific FIDO key.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 1-1-testuser-171
new-display-name Active
```

5. **De-register** (delete) a specific key using SOAP and HMAC authorization.

```
java -jar skfsclient.jar D https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 1-1-testuser-171
```

6. **Ping** the FIDO2 Server using SOAP and HMAC authorization.

```
java -jar skfsclient.jar P https://[FQDN]:8181 1 REST HMAC
162a5684336fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4
```

3.4.5—Perform Example Tests (SOAP and PASSWORD)

1. **Register a new FIDO key** to a user named *johndoe* using SOAP and PASSWORD authorization.

```
java -jar skfsclient.jar R https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe https://[FQDN]:8181
```

2. **Authenticate** the same user with the FIDO key registered in the step above using SOAP and PASSWORD authorization. Provide an authentication counter starting at 1, since this is the first authentication.

```
java -jar skfsclient.jar A https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe https://[FQDN]:8181 1
```

3. **Retrieve the information** about the list of FIDO keys registered by a specific user using SOAP and PASSWORD authorization.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! johndoe
```

4. **Update key information** for a specific key using SOAP and PASSWORD authorization. In this call you will be able to set the display name and deactivate or reactivate a specific FIDO key.

```
java -jar skfsclient.jar G https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! 1-1-testuser-171 new-display-name Active
```

5. **De-register** (delete) a specific key using SOAP and PASSWORD authorization.

```
java -jar skfsclient.jar D https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234! 1-1-testuser-171
```

6. **Ping** the FIDO2 Server using SOAP and PASSWORD authorization.

```
java -jar skfsclient.jar P https://[FQDN]:8181 1 REST PASSWORD
svcfidouser Abcd1234!
```

3.4.6—Outputs

Following are examples of successful outputs; this example uses REST and HMAC.

3.4.6.1—Registration

strongkey@fido2demo:~/skfsclient

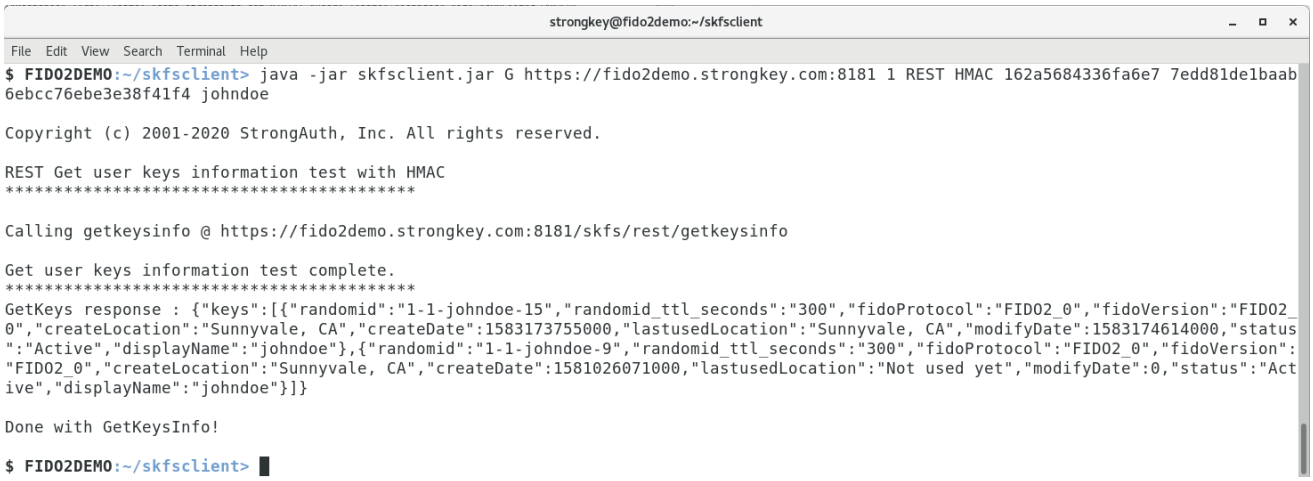
File Edit View Search Terminal Help

```
$ FIDO2DEMO:~/skfsclient> java -jar skfsclient.jar R https://fido2demo.strongkey.com:8181 1 REST HMAC 162a568433
6fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://fido2demo.strongkey.com:8181

Copyright (c) 2001-2020 StrongAuth, Inc. All rights reserved.

REST Registration test with HMAC
*******************************

Calling preregister @ https://fido2demo.strongkey.com:8181/skfs/rest/preregister
 Response : {"Response":{"rp":{"name":"demo.strongauth.com:8181"},"user":{"name":"johndoe","id":"c9DCjjmAd_wU6k7
WedAU4P2KcQfYgRHHttf0fzZXk1Y","displayName":"johndoe"},"challenge":"eF7pWiIyTOsz74iXgvt0Ug","pubKeyCredParams":[
{"type":"public-key","alg":-7},{"type":"public-key","alg":-35},{"type":"public-key","alg":-36},{"type":"public-k
ey","alg":-8},{"type":"public-key","alg":-43},{"type":"public-key","alg":-65535},{"type":"public-key","alg":-257
},{"type":"public-key","alg":-258},{"type":"public-key","alg":-259},{"type":"public-key","alg":-37},{"type":"pub
lic-key","alg":-38},{"type":"public-key","alg":-39}],"excludeCredentials":[{"type":"public-key","id":"yVNP5_KmXu
Cjk-g-36V75oT1dz52LLVAdropU0NJPt4TMmE4tDxnmiHbeNDXrB3kmYnOPr-eBUJdxhV9mRH0UiKNkq1lFIBys--G4s_OGzTZONsgbwSURElf06
6BnUuTdI79c3bG2ndhb_pHtSjhS89GiQNi40U8DAI6IiYHiqJz9wGRSUQdutziWOIbk1UeIvNpV4fm4w02A69tBKSPgm2I2N1e9IKys5Ml1WqCom
M","alg":-7}],"attestation":"direct"}}

Pre-Registration Complete.

Generating Registration response...

Simultor Response :
        id = e20kbABZPCPONZcpi0W7rAZ86YwHIgi8rf6VKiT-iFjQYfZlJDzqgOlmn_R4GkdfHE4wGHwSEsoUJVastGy5V4iWLdrV1TUVZ7_
c99QYA8ImgO14A9y09xuDcmTreh-oCbiYNkFis1k9fmw4CYm1OPAnKlYxWnudC2Sltqo_LC_Tggm-pdJkjOrDNeQydfxDH-S5uOIKMZFMd2Qfkvd
xVBmpQPSFoUNn09S-YsyoND8
        rawId = e20kbABZPCPONZcpi0W7rAZ86YwHIgi8rf6VKiT-iFjQYfZlJDzqgOlmn_R4GkdfHE4wGHwSEsoUJVastGy5V4iWLdrV1TUV
Z7_c99QYA8ImgO14A9y09xuDcmTreh-oCbiYNkFis1k9fmw4CYm1OPAnKlYxWnudC2Sltqo_LC_Tggm-pdJkjOrDNeQydfxDH-S5uOIKMZFMd2Qf
kvdxVBmpQPSFoUNn09S-YsyoND8
        response =      attestationObject = o2NmbXRmcGFja2VkZ2F0dFN0bXSjY2FsZyZjc2lnWEYwRAIge0uaXSD3usAv52r9pRTb
WoDyTpIRwHcuhA6gzHAsTzgCIAE1HSyqS3EqCH7thL4AcXx-qmV8RiRFDskuCLPishYvY3g1Y4FZAeQwggHgMIIBg6ADAgECAgRsK1jyMAwGCCqG
SM49BAMCBQAwZDELMAkGA1UEBhMCVVMxFzAVBgNVBAoTDlN0cm9uZ0F1dGggSW5jMSIwIAYDVQQLExlBdXRoZW50aWNhdG9yIEF0dGVzdGF0aW9u
MRgwFgYDVQQDDA9BdHRlc3RhdGlvbl9LZXkwHhcNMTkwNzE4MTcxMTI3WhcNMjkwNzE1MTcxMTI3WjBkMQswCQYDVQQGEwJVUzEXMBUGA1UEChMO
U3Ryb25nQXV0aCBJbmMxIjAgBgNVBAsTGUF1dGhlbnRpY2F0b3IgQXR0ZXN0YXRpb24xGDAWBgNVBAMMD0F0dGVzdGF0aW9uX0tleTBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABDH0hj6698S9n0dolffJpiY6pIhhDFLc6LcR3uLDjNcZHkLhForI5B4i7WErZAKCirbXQqPA0VTdOmyoAxktmYWj
ITAfMB0GA1UdDgQWBBQ0QtDgcEONNbOP0TRnvX0TgR4vGDAMBggqhkjOPQQDAgUAA0kAMEYCIQDtFtHY0K3IxDCLIYY4APLysMeM0U-Vkbwin2Sv
2IAqKwIhAKLdw0AWNMvTf6yTsLWWKPslDqY7uuF90Mx8NdKN6DC5aGF1dGhEYXRhWQE0ff249ru8royxQe-PAR1yMMAbP39S0SstHPXRt-Uc39pB
AAAAAAAAAAAAAAAAAAAAAAAAAAAsHttJGwAWTwjzjWXKYtFu6wGfOmMByIIvK3-lSok_ohY0GH2ZSQ86oDpZp_0eBpHXxxOMBh8EhLKFCVWrLRs
uVeIli3a1dU1FWe_3PfUGAPCJoDteAPctPcbg3Jk63ofqAm4mDZBYrNZPX5sOAmJtTjwJypWMVp7nQtkpbaqPywv04IJvqXSZIzqwzXkMnX8Qx_k
ubjiCjGRTHdkH5L3cVQZqUD0haFDZ9PUvmLMqDQ_pQECAyYgASFYIELoo42TYpk42WwG0IQClTyqUSMACTJOvICjX7tRWpsoIlggr5cE036437y8
xHcvMPsvXRDgdZXGYr7nlYpBO1C1Hi4
        clientDataJSON = eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY2hhbGxlbmdlIjoiZUY3cFdpSXlUT3N6NzRpWGd2dDBVZyIsIm9
yaWdpbiI6Imh0dHBzOi8vZmlkbzJkZW1vLnN0cm9uZ2tleS5jb206ODE4MSJ9
        type = public-key

Finished Generating Registration Response.

Registering ...

Calling register @ https://fido2demo.strongkey.com:8181/skfs/rest/register
 Response   : {"Response":"Successfully processed registration response"}

 Registration Complete.
*******************************

Done with Register!

$ FIDO2DEMO:~/skfsclient>
```

3.4.6.2—Authentication

```
strongkey@fido2demo:~/skfsclient
File  Edit  View  Search  Terminal  Help
$ FIDO2DEMO:~/skfsclient> java -jar skfsclient.jar R https://fido2demo.strongkey.com:8181 1 REST HMAC 162a568433
6fa6e7 7edd81de1baab6ebcc76ebe3e38f41f4 johndoe https://fido2demo.strongkey.com:8181

Copyright (c) 2001-2020 StrongAuth, Inc. All rights reserved.

REST Registration test with HMAC
*******************************

Calling preregister @ https://fido2demo.strongkey.com:8181/skfs/rest/preregister
 Response : {"Response":{"rp":{"name":"demo.strongauth.com:8181"},"user":{"name":"johndoe","id":"c9DCjjmAd_wU6k7
WedAU4P2KcQfYgRHHttf0fzZXk1Y","displayName":"johndoe"},"challenge":"eF7pWiIyTOsz74iXgvt0Ug","pubKeyCredParams":[
{"type":"public-key","alg":-7},{"type":"public-key","alg":-35},{"type":"public-key","alg":-36},{"type":"public-k
ey","alg":-8},{"type":"public-key","alg":-43},{"type":"public-key","alg":-65535},{"type":"public-key","alg":-257
},{"type":"public-key","alg":-258},{"type":"public-key","alg":-259},{"type":"public-key","alg":-37},{"type":"pub
lic-key","alg":-38},{"type":"public-key","alg":-39}],"excludeCredentials":[{"type":"public-key","id":"yVNP5_KmXu
Cjk-g-36V75oT1dz52LLVAdropU0NJPt4TMmE4tDxnmiHbeNDXrB3kmYnOPr-eBUJdxhV9mRH0UiKNkq1lFIBys--G4s_OGzTZONsgbwSURElf06
6BnUuTdI79c3bG2ndhb_pHtSjhS89GiQNi40U8DAI6IiYHiqJz9wGRSUQdutziWOIbk1UeIvNpV4fm4w02A69tBKSPgm2I2N1e9IKys5Ml1WqCom
M","alg":-7}],"attestation":"direct"}}

Pre-Registration Complete.

Generating Registration response...

Simultor Response :
        id = e20kbABZPCPONZcpi0W7rAZ86YwHIgi8rf6VKiT-iFjQYfZlJDzqgOlmn_R4GkdfHE4wGHwSEsoUJVastGy5V4iWLdrV1TUVZ7_
c99QYA8ImgO14A9y09xuDcmTreh-oCbiYNkFis1k9fmw4CYm1OPAnKlYxWnudC2Sltqo_LC_Tggm-pdJkjOrDNeQydfxDH-S5uOIKMZFMd2Qfkvd
xVBmpQPSFoUNn09S-YsyoND8
        rawId = e20kbABZPCPONZcpi0W7rAZ86YwHIgi8rf6VKiT-iFjQYfZlJDzqgOlmn_R4GkdfHE4wGHwSEsoUJVastGy5V4iWLdrV1TUV
Z7_c99QYA8ImgO14A9y09xuDcmTreh-oCbiYNkFis1k9fmw4CYm1OPAnKlYxWnudC2Sltqo_LC_Tggm-pdJkjOrDNeQydfxDH-S5uOIKMZFMd2Qf
kvdxVBmpQPSFoUNn09S-YsyoND8
        response =      attestationObject = o2NmbXRmcGFja2VkZ2F0dFN0bXSjY2FsZyZjc2lnWEYwRAIge0uaXSD3usAv52r9pRTb
WoDyTpIRwHcuhA6gzHAsTzgCIAE1HSyqS3EqCH7thL4AcXx-qmV8RiRFDskuCLPishYvY3g1Y4FZAeQwggHgMIIBg6ADAgECAgRsK1jyMAwGCCqG
SM49BAMCBQAwZDELMAkGA1UEBhMCVVMxFzAVBgNVBAoTDlN0cm9uZ0F1dGggSW5jMSIwIAYDVQQLExlBdXRoZW50aWNhdG9yIEF0dGVzdGF0aW9u
MRgwFgYDVQQDDA9BdHRlc3RhdGlvbl9LZXkwHhcNMTkwNzE4MTcxMTI3WhcNMjkwNzE1MTcxMTI3WjBkMQswCQYDVQQGEwJVUzEXMBUGA1UEChMO
U3Ryb25nQXV0aCBJbmMxIjAgBgNVBAsTGUF1dGhlbnRpY2F0b3IgQXR0ZXN0YXRpb24xGDAWBgNVBAMMD0F0dGVzdGF0aW9uX0tleTBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABDH0hj6698S9n0dolffJpiY6pIhhDFLc6LcR3uLDjNcZHkLhForI5B4i7WErZAKCirbXQqPA0VTdOmyoAxktmYWj
ITAfMB0GA1UdDgQWBBQ0QtDgcEONNbOP0TRnvX0TgR4vGDAMBggqhkjOPQQDAgUAA0kAMEYCIQDtFtHY0K3IxDCLIYY4APLysMeM0U-Vkbwin2Sv
2IAqKwIhAKLdw0AWNMvTf6yTsLWWKPslDqY7uuF90Mx8NdKN6DC5aGF1dGhEYXRhWQE0ff249ru8royxQe-PAR1yMMAbP39S0SstHPXRt-Uc39pB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAsHttJGwAWTwjzjWXKYtFu6wGfOmMByIIvK3-lSok_ohY0GH2ZSQ86oDpZp_0eBpHXxxOMBh8EhLKFCVWrLRs
uVeIli3a1dU1FWe_3PfUGAPCJoDteAPctPcbg3Jk63ofqAm4mDZBYrNZPX5sOAmJtTjwJypWMVp7nQtkpbaqPywv04IJvqXSZIzqwzXkMnX8Qx_k
ubjiCjGRTHdkH5L3cVQZqUD0haFDZ9PUvmLMqDQ_pQECAyYgASFYIELoo42TYpk42WwG0IQClTyqUSMACTJOvICjX7tRWpsoIlggr5cE036437y8
xHcvMPsvXRDgdZXGYr7nlYpBO1C1Hi4
        clientDataJSON = eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY2hhbGxlbmdlIjoiZUY3cFdpSXlUT3N6NzRpWGd2dDBVZyIsIm9
yaWdpbiI6Imh0dHBzOi8vZmlkbzJkZW1vLnN0cm9uZ2tleS5jb206ODE4MSJ9
        type = public-key

Finished Generating Registration Response.

Registering ...

Calling register @ https://fido2demo.strongkey.com:8181/skfs/rest/register
 Response   : {"Response":"Successfully processed registration response"}

 Registration Complete.
*******************************

Done with Register!

$ FIDO2DEMO:~/skfsclient>
```

3.4.6.3—List FIDO2 Keys



3.4.6.4—Update Information of Keys

3.4.6.5—De-register/Delete a Key

3.4.6.6—Ping StrongKey FIDO2 Server

3.4.7—Policy *skfsclient* Examples

3.4.7.1—Get Policy

Get the contents of a specific policy returned as a base64-encoded JSON object. Policies are specified by their *sid* and *pid*.

Example:

```
java -jar /usr/local/strongkey/skfsclient/skfsclient.jar GP
https://example.strongkey.com:8181 1 REST PASSWORD fidoadminuser Abcd1234!
False 1 1
```

3.4.7.2—Patch Policy

Update an existing policy. Policies are specified by their *sid* and *pid*.

Example:

```
java -jar /usr/local/strongkey/skfsclient/skfsclient.jar PP
https://example.strongkey.com:8181 1 REST HMAC 162a5684336fa6e7
7edd81de1baab6ebcc76ebe3e38f41f4 1 1 Active "" '{"FidoPolicy": {"name":
"DefaultPolicy UPDATED","copyright": "AAAA","version": "1.0","startDate":
"1460103870871","endDate": "1760103870871","system": {"requireCounter":
"mandatory","integritySignatures": false,"userVerification": ["required",
"preferred", "discouraged"],"userPresenceTimeout": 0,"allowedAaguids":
["all"],"algorithms": {"curves": ["secp256r1", "secp384r1", "secp521r1",
"curve25519"],"rsa": ["rsassa-pkcs1-v1_5-sha1", "rsassa-pkcs1-v1_5-sha256",
"rsassa-pkcs1-v1_5-sha384", "rsassa-pkcs1-v1_5-sha512", "rsassa-pss-sha256",
"rsassa-pss-sha384", "rsassa-pss-sha512"],"signatures": ["ecdsa-p256-sha256",
"ecdsa-p384-sha384", "ecdsa-p521-sha512", "eddsa", "ecdsa-p256k-
sha256"]},"attestation": {"conveyance": ["none", "indirect", "direct",
"enterprise"],"formats": ["fido-u2f", "packed", "tpm", "android-key",
"android-safetynet", "none"]},"registration": {"displayName":
"required","attachment": ["platform", "cross-platform"],"residentKey":
["required", "preferred", "discouraged"],"excludeCredentials":
```

```
"enabled"},"authentication": {"allowCredentials": "enabled"},"authorization":
{"maxdataLength": 256,"preserve": true},"rp": {"name": "FIDO Server","id":
"strongkey.com"}},"extensions": {"example.extension": true}}}'
```

3.4.7.3—Create Policy

Create a new policy. The result of this command will return an *sid-pid* pair that can be used in to reference this policy.

Example:

```
java -jar /usr/local/strongkey/skfsclient/skfsclient.jar CP
https://example.strongkey.com:8181 1 REST HMAC 162a5684336fa6e7
7edd81de1baab6ebcc76ebe3e38f41f4 Active "" '{"FidoPolicy": {"name":
"DefaultPolicy2","copyright": "AAAA","version": "1.0","startDate":
"1460103870871","endDate": "1760103870871","system": {"requireCounter":
"mandatory","integritySignatures": false,"userVerification": ["required",
"preferred", "discouraged"],"userPresenceTimeout": 0,"allowedAaguids":
["all"],"algorithms": {"curves": ["secp256r1", "secp384r1", "secp521r1",
"curve25519"],"rsa": ["rsassa-pkcs1-v1_5-sha1", "rsassa-pkcs1-v1_5-sha256",
"rsassa-pkcs1-v1_5-sha384", "rsassa-pkcs1-v1_5-sha512", "rsassa-pss-sha256",
"rsassa-pss-sha384", "rsassa-pss-sha512"],"signatures": ["ecdsa-p256-sha256",
"ecdsa-p384-sha384", "ecdsa-p521-sha512", "eddsa", "ecdsa-p256k-
sha256"]},"attestation": {"conveyance": ["none", "indirect", "direct",
"enterprise"],"formats": ["fido-u2f", "packed", "tpm", "android-key",
"android-safetynet", "none"]},"registration": {"displayName":
"required","attachment": ["platform", "cross-platform"],"residentKey":
["required", "preferred", "discouraged"],"excludeCredentials":
"enabled"},"authentication": {"allowCredentials": "enabled"},"authorization":
{"maxdataLength": 256,"preserve": true},"rp": {"name": "FIDO Server","id":
"strongkey.com"}},"extensions": {"example.extension": true}}}'
```

3.4.7.4—Delete Policy

Delete a specific policy. Policies are specified by their *sid* and *pid*.

Example:

```
java -jar /usr/local/strongkey/skfsclient/skfsclient.jar DP
https://example.strongkey.com:8181 1 REST PASSWORD fidoadminuser Abcd1234! 1 2
```

3.5—Example JSON

Following is an example JSON, the properties of which are defined in this document.

```
{
  "FidoPolicy": {
    "name": "DefaultPolicy",
    "copyright": "",
    "version": "1.0",
    "startDate": "1606957205",
    "endDate": "1760103870871",
    "system": {
      "requireCounter": "mandatory",
      "integritySignatures": false,
      "userVerification": ["required", "preferred", "discouraged"],
      "userPresenceTimeout": 0,
      "allowedAaguids": ["all"], "jwtKeyValidity": 365, "jwtRenewalWindow":
30},
"algorithms": {
        "curves": ["secp256r1", "secp384r1", "secp521r1", "curve25519"],
        "rsa": ["rsassa-pkcs1-v1_5-sha256", "rsassa-pkcs1-v1_5-sha384",
"rsassa-pkcs1-v1_5-sha512", "rsassa-pss-sha256", "rsassa-pss-sha384",
"rsassa-pss-sha512"],
        "signatures": ["ecdsa-p256-sha256", "ecdsa-p384-sha384", "ecdsa-p521-
sha512", "eddsa", "ecdsa-p256k-sha256 "]
      },
      "attestation": {
        "conveyance": ["none", "indirect", "direct", "enterprise"],
        "formats": ["fido-u2f", "packed", "tpm", "android-key", "android-
safetynet", "none"]
      },
      "registration": {
        "displayName": "required",
        "attachment": ["platform", "cross-platform"],
        "residentKey": ["required", "preferred", "discouraged"],
        "excludeCredentials": "enabled"
      },
      "authentication": {
        "allowCredentials": "enabled"
      },
      "authorization": {
        "maxdataLength": 256,
        "preserve": true
      },
      "rp": {
        "name": "FIDOServer",
        "id": "strongkey.com"
      },
      ,
    "extensions": {
      "example.extension": true
    },
      "jwt": { "algorithms": ["ES256", "ES384", "ES521"], "duration": 30,
```

```
"required": ["rpid","iat", "exp", "cip", "uname", "agent"], "signingCerts": {
"DN": "CN=StrongKey Key Appliance,O=StrongKey", "certsPerServer": 2 } } }
  }
}
```

3.6—FIDO2 Policy JSON

3.6.1—Terms

3.6.1.1—FIDO2 Authenticator

Hardware or software used to generate user credentials and prove user identity.

- Roaming Authenticator: FIDO2 Authenticators that can be used on multiple devices. It connects to the user's device via USB, NFC, or Bluetooth. Formats include USB sticks and BLE-enabled smart phones. Sometimes roaming Authenticators are referred to as security keys.
- Platform Authenticator: FIDO2 Authenticator built into the user's device. These Authenticators can come in the form of the fingerprint readers built into smart devices and features such as Windows Hello on a laptop.

3.6.1.2—Relying Party (RP)

The website or application relying on FIDO2 to authenticate the identity of the user; the website or application to which FIDO2 is being used to login.

3.6.1.3—StrongKey FIDO2 Server (SKFS)

FIDO2 Server is a separate server with which the RP communicates to manage all FIDO2 transactions and data.

3.6.1.4—Client Device

The device with which the user is connecting to the RP and the Authenticator. This can often be the computer or mobile device used while conducting FIDO2 operations.

3.6.1.5—Client Platform

The application used to interface with the RP and the Authenticator. This can often be the browser or mobile application used while conducting FIDO2 operations.

3.6.1.6—Credential

A set of data used by the Authenticator and verified by SKFS to prove user identity.

3.6.2—Policy Options

3.6.2.1—General Configuration

- name: The name of the FIDO2 policy
- copyright: A plain language name of the copyright
- version: The policy format version

- **startDate**: The policy effective start date in milliseconds; when the FIDO2 Server should start using this policy
- **endDate**: The policy effective end date in milliseconds; when the FIDO2 Server should stop using this policy

3.6.2.2—system

3.6.2.2.1—requireCounter

Does the Authenticator need to use a signature counter? A signature counter is a number the Authenticator stores and is increased by some positive value every time the Authenticator is used to authenticate. The purpose of this feature is to help SKFS in detecting clone Authenticators. It does this by storing its own instance of the signature counter and compares it to the Authenticator's signatures counter upon any authentication action. The Authenticator's counter must be higher than SKFS's counter; otherwise a cloned Authenticator may have been used.

Allowed values:

- **mandatory**: The Authenticator must have a counter. This will guarantee the added security of having a counter but it will restrict the number of Authenticator models that can be used. By definition most Authenticator should support using a counter so this option will still allow a majority of Authenticators models.
- **optional**: Allowed to not have a counter, but not required. This option will not restrict the breadth of accepted Authenticator models in any way. It will allow both Authenticators that support counters and those that do not. If an Authenticator supports a signature counter than one will be used.

3.6.2.2.2—integritySignatures

Currently not implemented.

An integrity signature is the digital signature return during the registration process as part of the attestation. It provides evidence of the authenticity of registration transaction. is registration signature stored.

Allowed values:

- **true**
- **false**

3.6.2.2.3—userVerification

User verification is the process by which the Authenticator confirms the user's identity locally before authorizing the use of Authenticator functionality. This confirmation can come in to form various authorization gestures; some examples are having the user interact with a fingerprint reader, enter a PIN or password, or use facial recognition. User verification is not necessary but it acts as a secondary layer of security to make sure the Authenticator is not being used by a someone other than the intended user. Allowed values:

- **required**: Must have user verification. This means that if an Authenticator is not able to perform user verification then operations will be rejected by the FIDO2 Server. The benefit of this option is the added security it brings to the FIDO2 operations. By only selecting this option, you can guarantee that all users of the RP have this added level of security, making their transaction potentially more trustworthy than those with Authenticators without user verification. The downsides to picking this option are that it will cause SKFS to reject operations from Authenticators without user verification, potentially reducing the user base; also, by forcing the user verification process it will slow the user experience.
- **preferred**: Verify users if possible; otherwise don't. If user verification is possible then the Authenticator must use it. If user verification is not possible then the Authenticator can perform the operation without user verification without issue. Including this option has all Authenticators that can use user verification do so without outright rejecting Authenticators that can't use user verification. This gives the potentially increased security of user verification without the negatives of possibly rejecting a user's Authenticator of choice and impairing the user experience.
- **discouraged**: Don't verify users. Even if user verification is possible the Authenticator should not use user verification. The advantage of this option is that it can improve the ease of the user experience. Instead of potentially having to input a PIN or password, the user might be able to bypass this step if the Authenticator allows it.

3.6.2.2.4—userPresenceTimeout

Currently not implemented in WebAuthn.

Time in seconds that user presence will remain valid without asking for proof of user presence. The purpose of this option is to allow a more fluid end user experience. By default, user presence must be checked every time a user is required to authenticate. This can become a problem if a process requires the user to authenticate or authorize multiple times, making the process more work for the user. By setting *userPresenceTimeout* to a number greater than 0, it allows the Authenticator to not have to check user presence before authenticating/authorizing multiple times on behalf of the user. The trade off for increasing the time for *userPresenceTimeout* is that it opens up the possibility that the user is not actually present at the time of the transaction. This can happen if the Authenticator is built into the client device or the roaming Authenticator was left in a client device, and the user is away. This window of time allows the possibility that either a bad actor or automatic process occurs without the user's consent, but the Authenticator will still make it appear the user gave consent by going through with this transaction without them.

3.6.2.2.5—allowedAaguids

If *aaguids* is set then attestation formats must be truncated only include *packed* and *tpm*.

Aaguids are unique Authenticator model identifiers implemented by the Authenticator's manufacturer. A manufacturer will create an *aaguid* for each model of Authenticator they produce so that the Authenticator's unique properties can be easily confirmed. By default SKFS accepts all *aaguids*. This option enables restriction of the specific models of Authenticators SKFS will accept by specifying the model's *aaguids*. The advantage of restricting the accepted Authenticator models is it can allow an added layer of standardization and security. If a company distributes only one model of Authenticator to all their employees to sign in to an internal website, they can restrict SKFS to only allow that Authenticator's *aaguid*. Then if any non-employee tries registering without a valid Authenticator, they will be automatically rejected and that irregularity will be logged in SKFS. Currently only two attestation formats pass the *aaguid* during the registration process: *packed* and *tpm*. This is why if *aaguids* are specified, then attestation formats should only contain *packed* and *tpm* formats.

Allowed Values:

- all: Accept all Authenticators regardless of *aaguid* status. This will no restrict the use of Authenticators based on make and model. The benefit of this options is that it allows for the greatest variety of Authenticators. An issue with this option is that it will it is completely unrestricted. By allowing all Authenticators (that match other option criteria), it might make it easier for a potential bad actor to access the RP since they are able to use which ever Authenticators they currently have instead of needing to have the same Authenticator model as required by SKFS.
- specify specific *aaguids* to accept: SKFS rejects all Authenticators that are not the correct make and model specified by *aaguids*. This can make it harder for potential bad actors to access the RP by automatically rejecting all Authenticators the individual has that are not the exact Authenticator model specified. Another benefit is that you can guarantee certain security features are available in the Authenticators that are accepted by SKFS by only specifying models that have those particular features.

3.6.2.2.6—jwtKeyValidity

The JWT Key is in reference to a cryptographic key that is used to sign the JWTs produced by the server. For more information on JWTs please reference 1.6.4. JWTKeyValidity indicates the days, the cryptographic keys are valid before they must be replaced. By default this value is set to 365 days or 1 year.

3.6.2.2.7—JwtRenewalWindow

The window of time that the admin has to renew the JWT signing certificates. By default the value is set to 30 days. Within these 30 days before the JWT signing certificate expires the admin must create new certificates from the JWT Certificate Authority and import them into the JWT signing KeyStore and JWT signing TrustStores.

3.6.2.2.8—algorithms

Currently *Elliptic Curve (EC)* is preferred over *Rivest Shamir Adelman (RSA)* in code.

This section involves the encryption algorithms used for digital signature creation and verification. The reason why it might be beneficial to restrain the allowed algorithms is to maintain a desired level of encryption strength. The more sensitive or the more entities depend on secure encryption of a particular set of data then generally the higher the bit size and/or the stronger the algorithm used. So depending on how vital user authentication is to the RP the chosen algorithms should be picked.

curves (EC)

The allowed curves of the signature used by the Authenticator used during registration. Allowed values:

- **all**: All of the following algorithms
- **secp256r1**
- **secp384r1**
- **secp521r1**
- **curve25519**
- **none**: None of the above algorithms

rsa

The allowed RSA algorithms for the digital signature used by the Authenticator during registration. The *rsa* and *signatures* options pertain to the same functionality; the signing of the responses sent return by the Authenticator. By design all EC algorithms are prioritized over *rsa* if both are set. Accepting RSA or EC algorithms can be avoided by using *none* as the desired option. If *rsa* has a value of *none* then EC must have one or more accepted algorithms, and vice versa. Allowed values:

- **all**: All of the following algorithms
- **rsassa-pkcs1-v1_5-sha1**
- **rsassa-pkcs1-v1_5-sha256**
- **rsassa-pkcs1-v1_5-sha384**
- **rsassa-pkcs1-v1_5-sha512**
- **rsassa-pss-sha256**
- **rsassa-pss-sha384**
- **rsassa-pss-sha512**
- **none**: None of the above algorithms

signatures (EC)

The allowed EC algorithms for the digital signature used by the Authenticator during registration. Allowed values:

- **all**: All of the following algorithms
- **ecdsa-p256-sha256**
- **ecdsa-p384-sha384**
- **ecdsa-p521-sha512**
- **eddsa**
- **ecdsa-p256k-sha256**
- **none**: None of the above algorithms

3.6.2.2.9—attestation

An attestation is a data object that is sent can be sent by the Authenticator by the during registration. Most Authenticators should pass an attestation. The attestation contains data about the Authenticator, a digital signal, the generated cryptographic key, and more. The attestation can come in a a verity of forms based on Authenticator capabilities and client platform.

conveyance

Attestation conveyance is the way in which SKFS requires the attestation to be created. Any combination of the following may be chosen. It might beneficial to only allow a subset of conveyances to guarantee a certain level of security or to guarantee such functionality and maintain anonymity. By having all conveyances present in this option it allows for the largest number of possible registration options, but sacrifices control over the registration process.

Allowed values:

- **none**: SKFS does not want an attestation to be sent. By having this conveyance present it signifies that SKFS will accept if the RP does not return an attestation during registration. This option nullifies the security given by the attestation, so it is not recommended unless the alternative registration response format has been thoroughly understood and trusted.
- **indirect**: SKFS prefers receiving Authenticator generated attestation statement but can use anonymization CA generated attestation statement. By allowing this option it gives the RP the ability to generate the attestation while maintaining the user's anonymity. This could be useful if maintaining the privacy of personal data is of top priority.
- **direct**: SKFS expects to receive an attestation generated by the Authenticator. This is the most recommended option because it ensures all the expected functionality of the attestation. Since the attestation is created directly by the Authenticator and then processed by SKFS, only two parties must be trusted. This is different from the indirect option. Using an indirect conveyance requires the process to trust the RP and a separate Certificate Authority along with the authenticator and FIDO2 Server; with the direct attestation it ensures that all expected FIDO2 functionality is preserved.
- **enterprise**: Signifies the use of uniquely identifying information in the attestation. This option is intended for use in a controlled deployment within an organization to tie registration to specific authenticators.

formats

An attestation's format defines what data is contained in the attestation and how that data is organized. The different formats are designed to work with specific types of Authenticators, clients, and client platforms. Which format is used during registration is decided by the client platform (the application or browser directly interacting with the Authenticator). All or any subsection of the following formats can be set to this option which will force SKFS to only accept the attestation formats specified. Selecting only a subset can allow only specific platforms to authenticate or to handle the case that specific *aaguids* have been specified in the *allowedAaguids* option.

Allowed values:

- **packed**: The default modern WebAuthn compatible format. All of the following formats exists to fit specific Authenticator or client hardware specifications. Packed is the most versatile of attestation formats when it come for attestation types. It supports Basic, Self, AttCA types. Packed is also one of two formats that passes an Authenticator's *aaguid* during registration along with tpm format. This means that if specific *aaguids* are specified in the *allowedAaguids* option then this format and/or *tpm* format must be specified.
- **tpm**: Used by Authenticators that have a Trusted Platform Module for cryptographic functionality. The built-in use of a TPM is an added level of security that other Authenticators might not have. A TPM is a computer chip whose dedicated purpose is to securely store artifacts such as passwords, certificates, and encryption keys. All three of these artifacts can potentially be used within FIDO2, depending on the Authenticator. Specifying *tpm* format allows the use of Authenticators with this added security feature to be used properly by the client platform. *tpm* is also one of two formats that passes an Authenticator's *aaguid* during registration along with packed format. This means if *aaguids are* specified in the *allowedAaguids* option then *tpm* format and/or packed format must be specified. Supports *AttCA* and *ECDAA* attestation types.
- **android-key**: Used when the client platform is Android. This format is based on Android key attestation, which is a native Android protocol for the creation, storage, and use of the cryptographic keys. By specifying this format Android devices that support Android key attestation will be able to perform operations against SKFS. Supports the *basic* attestation type.
- **android-safetynet**: Used when the Authenticator is a platform Authenticator based on an Android platform whose attestation statement is based on the SafetyNet API. SafetyNet is a service whose purpose is to detect system abuse to help determine if servers and applications are running on a genuine Android device. By specifying this option it allows SKFS to handle registration operations from devices that support this added device integrity check, making the operation all that more trustworthy. Supports the *basic* attestation type.

- fido-u2f: Used for U2F Authenticators. FIDO U2F Authenticators are legacy Authenticators that follow FIDO U2F protocol, a predecessor to modern FIDO2. Allowing this format will allowing these older Authenticators to be used. Also, since the client platform can define what attestation format will be used it is possible that the client platform might use *fido-u2f* even with a FIDO2 Authenticator. Supports *basic* and *AttCA* attestation types.
- none: Used when the Relying Party does not wish to receive an attestation.

3.6.2.2.10—registration

displayName

The *displayName* is a plain language name used to identify the Authenticator to the user. This name will be displayed in case the user wishes to manage multiple Authenticators associated with their account. The *displayName* is also sent to the client platform during registration to identify the user.

Possible values:

- required: A display name is required. This means that when the user goes through the registration process both a username and a *displayName* will be required by SKFS for the *preregister* request. A typical implementation will involve having the user input a username and display name during the registration process. An advantage to having display name required is that it allows a user to easily manage multiple Authenticators per account. This is thanks to having an easily readable display name associated with each Authenticator on their account so that the keys can be managed.
- preferred: If a display name is provided by the user then *displayName* will be used. If a display name is not provided then the username will be used as the *displayName*. Providing this option allows for different FIDO2 implementations on the RP's side while working with a single SKFS. One RP could have multiple Authenticators per account implemented, which inherently involves a unique display name, while another only allows one Authenticator per account, therefore not needing the display name.
- none: The username will be used as *displayName*—a simpler registration process due to only requiring the username from the user. The negative to this option is that it only allows a user to register one Authenticator per username. This is not recommended; if the user loses that one Authenticator they will be unable to access the account.

attachment

The Authenticator attachment is the transport protocol used by the client platform to communicate with the Authenticator during operations.

Allowed values:

- platform: Used to communicate with platform Authenticators.
- cross-platform: Used to communicate with roaming Authenticators.

residentKey

A resident key is a data structure that can be stored by the client platform to be used in a Authenticator selection process during authentication. If supported by the client platform, it is possible for the client platform to display a list of options for available Authenticators the user wishes to use from the locally available registered Authenticators. Note that this client platform UI is not a required feature to enable the use of multiple Authenticators for one account. Allowed values:

- **required**: *residentKey* must be created. This means that the Authenticator must return a resident key. This might not be a possibility for roaming Authenticators; only allowing this option might result in rejected roaming Authenticators. This will greatly reduce the number of possible Authenticators that can be used with this SKFS.
- **preferred**: Create a *residentKey* if possible; otherwise don't. This will require any Authenticator that can generate a resident key to do so, while not outright rejecting Authenticators that don't support resident keys.
- **discouraged**: Don't create a *residentKey*. This means that even Authenticators that support resident keys will not generate a resident key during registration. This means that there will not be any client-stored FIDO key data but, consequently, no possible client platform UI for Authenticator selection. Note that this client platform UI is not a required feature to enable multiple Authenticators for one account.

excludeCredentials

When *excludeCredentials* is enabled, a list of credential identifying information of previously generated credentials is sent to the Authenticators during registration. The Authenticator will check if any of these credentials were generated independently; and if not, reject the operation. This avoids having an Authenticator needlessly create another credential for the same account and RP. A user with multiple credentials for the same account does not add any more functionality than having a signed credential; the user will use their username and Authenticator to authenticate themselves in the same fashion with a single credential associated with their account.

- **enabled**: Sends a list of information of credentials to avoid duplicate credential creation.
- **disabled**: Sends a list of information of credentials; potential for duplicate credential creation.

3.6.2.2.11—authentication

allowCredentials

When *allowCredentials* is enabled a list of identifying information for registered credentials is sent to the Authenticator upon authentication. The Authenticator uses this list of credentials to identify the credential to use for authenticating the user.

Possible values:

- **enabled**: Sends a list of credential identifying information to the Authenticator, allowing the Authenticator to identify the proper credential to use for authentication.
- **disabled**: Don't send a list of credential identifying information to the Authenticator and therefor allowing the Authenticator to rapidly identify the proper credential to use for authentication.

3.6.2.2.12—authorization

*Currently not implemented.

maxdataLength

The size of the data length in bytes. This value can be modified to most closely match the defined size of the transaction data to be handled and potentially stored.

Possible values:

- 0–256 bytes

preserve

Whether or not to preserve data of the authorized transaction.

Possible values:

- **true**: Preserve data of the authorized transaction. This will allow the possibility for future auditing of any transaction authorized through SKFS. A negative that each transaction will take up storage on the machine or cluster running SKFS.
- **false**: Don't preserve data of the authorized transaction. This will make the possibility for future auditing of of transactions authorized through SKFS not possible. A positive might be that transaction data will not longer take up storage on the machine or cluster running SKFS.

3.6.2.2.13—rp

Relying Party identifiers. The Relying Party is the website or application that relies on FIDO2 to authenticate users.

name

The plain language name of the Relying Party.

id

The RPID identifies the Relying Party. If an RPID is not set in the policy its default value will be the effective domain of the origin of the caller who calls the Authenticator. *id* can be set as the domain name of the Relying Party or the hostname of the Relying Party. If the domain

name is used such as *example.com*, an Authenticator registered on *app.example.com* can also authenticate on *app2.example.com*. If the hostname of the RP is used—for example *app.example.com*—then an Authenticator registered on *app.example.com* will only be able to authenticate on *app.example.com*.

3.6.3—Extensions

*Currently no extensions are implemented.

Extensions are ways in which the functionality of FIDO2 operations (*registration*/*authentication*) can be extended to suit particular needs.

3.6.4—JWT

JSON Web Token (JWT) is a defined structure used to represent claims to be handled between two parties. SKFS uses JWTs as confirmation that the user who received the JWT has been authenticated. See *Appendix C* for more details.

algorithms

This is a list of the possible Elliptic Curve algorithms that SKFS is authorized to use for signing the JWTs.

duration

Duration is how long the JWT is valid for after it is generated by SKFS. The units are in minutes.

required

A list of all the required content for the payload. This list is consulted any time a JWT is verified by SKFS to make sure the JWT has all the required information.

Allowed values:

- **rpid:** The relying party id
- **iat:** The start date and time for when the JWT is created
- **exp:** The end date and time for when the JWT expires
- **cip:** The client IP address
- **uname:** The users username
- **agent:** The User Agent used by the user when authorizing to receive the JWT

signingcerts

Options involving the JWT signing certificates.

Allowed values:

- **dn:** This is the distinguished name to identify the JWT signing certificates.
- **certsperserver:** This is how many certificates there are per server in the cluster. Each server uses its own unique certificates to sign the JWTs it produces.

4—Developers

Throughout this section, snippets of code will be displayed in different colors to help distinguish from each other the languages being used. These colors are as follows:

```
Cyan code is JavaScript directed to the FIDO2 Authenticator.
Green code is JavaScript directed to its own back-end REST web services.
Grey code is JavaScript processing.
Blue code is Java in the business web application.
Purple code is JSON data structure input.
Red code is JSON data structure output.
Orange code is HTML.
```

 **NOTE:** StrongKey's APIs are described in more detail in the StrongKey FIDO2 Server API section, 4.2.

4.1—FIDO2-enabling a Web Application

In FIDO2, it is the web application's (service provider application's) job to determine when to request a challenge for a FIDO2 workflow. At a bare minimum, the service provider must provide a method for new registrations and authentication (login) attempts.

For visualization purposes, these workflows will be associated with similar password-based workflows. This is not done to imply that FIDO2 workflows can be used as drop-in replacements for password workflows (e.g., a "change password" workflow does not have a clear FIDO2 equivalent), but rather to help in understanding the workflows.

Sample code is from StrongKey's Basic Java Sample Application.

4.1.1—Initial Registration

In a typical new password-based registration, a web application will provide a form asking for a unique identifier, usually a username and a password (twice). Upon receiving the username and password, the web application will verify that the username has not been already taken by another user. If it has not, the web application will typically pass the username and password to its authentication module that will salt and hash the password before storing the username and now-hashed password in a database.

In a new FIDO2 registration workflow, rather than storing a username and hashed password, the workflow seeks to store a username and cryptographic public key. However, FIDO2 introduces many new concepts that are best explained in detail. The steps to achieve this are listed on the following pages.

1. User submits identification information for the credential.

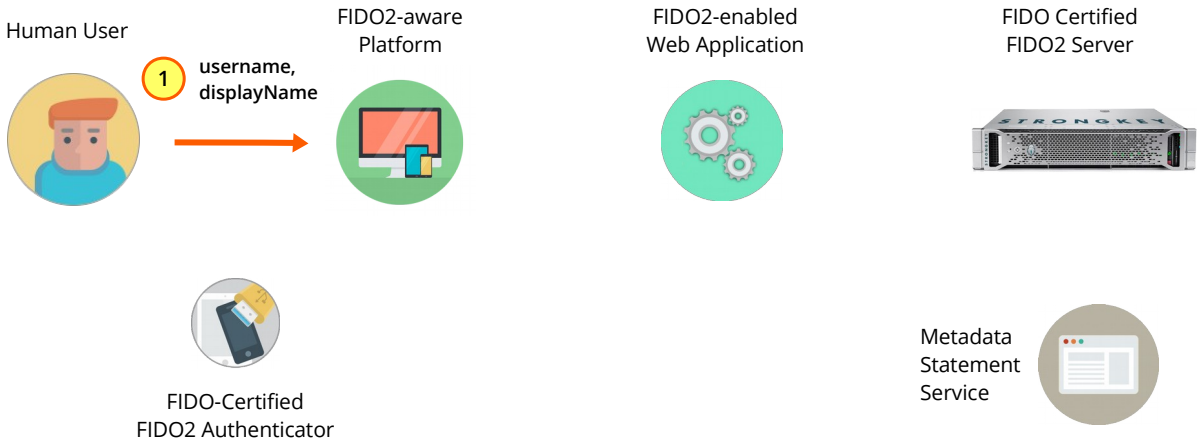


Figure 1—1: Registration: User supplies username and displayName to the web form.

In a new FIDO2 registration, a web application will provide a form asking for a *username* and a *displayName*. The *username* serves the same function as it does in the password-based workflow. The *displayName* is unrelated to security and is a means to set a human-readable identifier to the key pair being created. It is strongly recommended that web application developers use the string "Initial Registration" for the *displayName* and do not allow the user to modify this; every unique website will require the user to go through an "initial FIDO2 registration" process when enabling their FIDO2 Authenticator for that site, and it is unlikely that most users will forget their first FIDO2 Authenticator and the initial registrations at websites. When enabling modifications to the *displayName* in the web application, programmers might want to consider allowing users to only append to the "Initial Registration" string so as to preserve the context for the user even as they add many other FIDO2 Authenticators to a website. The HTML form used in the basic server demo is shown below.

```
<div id="regPanel" class="form-panel">
  <h2>Register</h2>
  <div id="regUsernamePanel">
    <input autocomplete="off" class="met" id="regUsername"
placeholder="Username" required="" type="text">
  </div>
  <div>
    <input autocomplete="off" class="met" id="regDisplayName"
placeholder="Initial Registration" required="" type="text">
  </div>
  <div>
    <button class="met" id="regSubmit">Register New Key</button>
  </div>
</div>
```

2. The JS front end sends *username* and *displayName* to the web application back end.

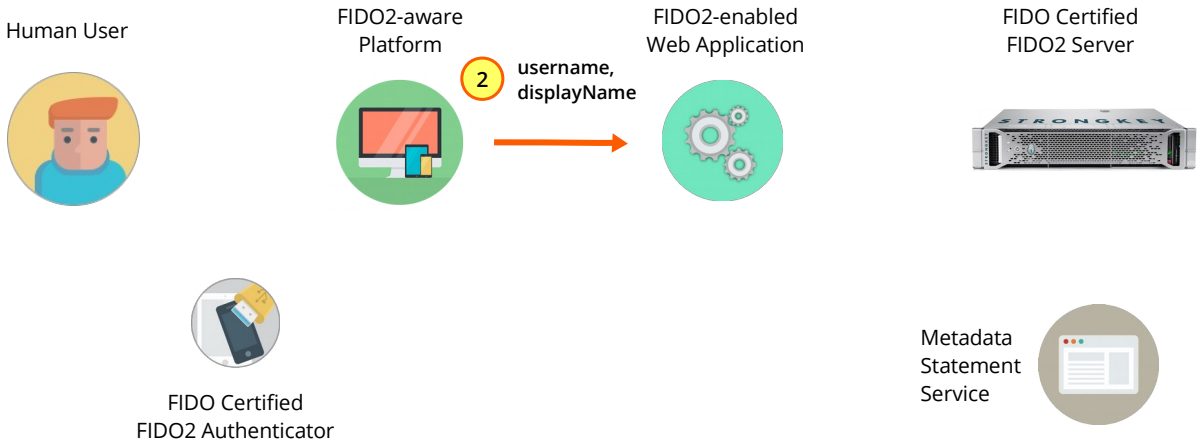


Figure 1–2: Registration: Form fields are submitted to web application back end.

Upon the user submitting the information to the web form, the service provider's web application JS front end validates the information and calls a web service—named 'preregister' in the JS code to align with the SKFS *preregister* web service—to handle the registration process.

```
this.post('preregister', {
'username': $('#regUsername').val(),
'displayName': $('#regDisplayName').val()
})
```

3. The submitted *username* is checked for uniqueness and FIDO2-enablement.

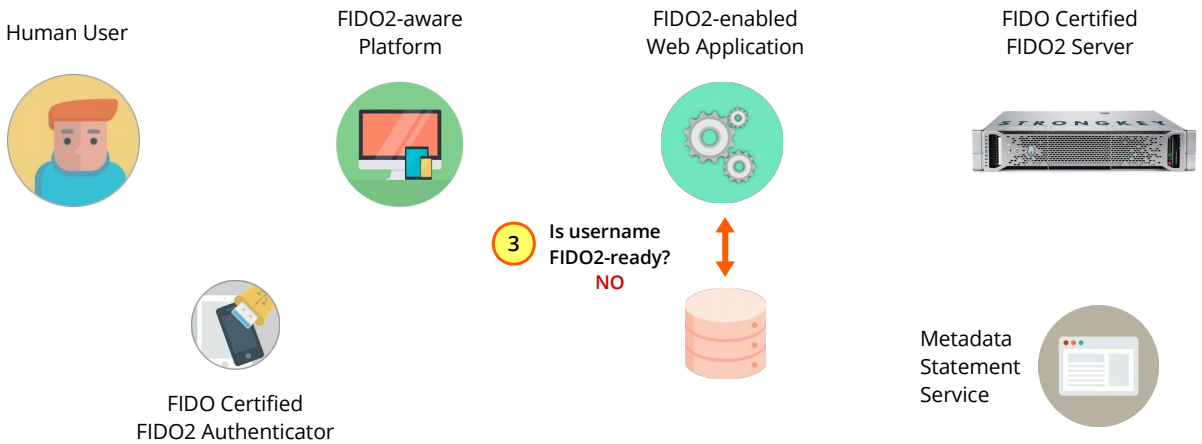


Figure 1–3: Registration: The web application ensures the username is unique; the user is NOT FIDO2-enabled.

Upon receiving the *username* and *displayName*, the service provider's web application back end verifies that the *username* is not already in use and that the user has NOT previously registered a FIDO2 key with this site (see Error: Reference source not found). If either verification fails to pass the test, the back-end application must respond appropriately to the user. If the *username* is not already in use and if the user has not previously registered a FIDO2 key, the service provider's web application proceeds with the next step.

```
String username = getValueFromInput(Constants.RP_JSON_KEY_USERNAME,
input);
String displayName = getValueFromInput(Constants.RP_JSON_KEY_DISPLAYNAME,
input);

//Verify User does not already exist
if (!doesAccountExists(username)){
  String prereg = SKFSClient.preregister(username, displayName);
  HttpSession session = request.getSession(true);
  session.setAttribute(Constants.SESSION_USERNAME, username);
  session.setAttribute(Constants.SESSION_ISAUTHENTICATED, false);
  session.setMaxInactiveInterval(Constants.SESSION_TIMEOUT_VALUE);
  return generateResponse(Response.Status.OK, prereg);
}
```

4. The web application sends a *preregister* web service request to SKFS.

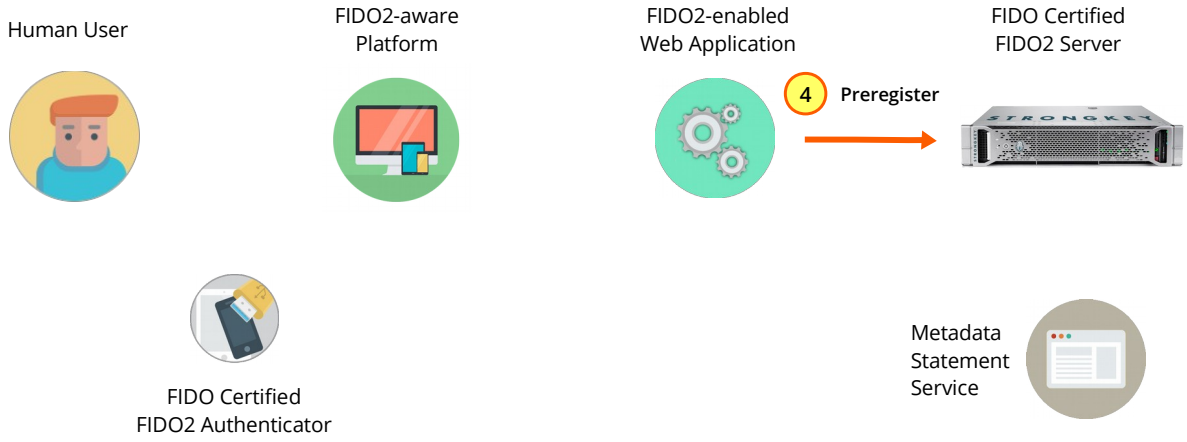


Figure 1—4: Registration: The web application sends a preregister web service request to SKFS.

Here is the JSON for the *preregister* input:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
```

```
  },
  "payload": {
    "username": "johndoe",
    "displayName": "Initial Registration",
    "options": {
      "attestation": "direct"
    },
    "extensions": "{}"
  }
}
```

The service provider's web application now sends a *preregister* web service request to SKFS, using the REST protocol, passing the *username* and *displayName* to SKFS via a POST request (see Error: Reference source not found). The Java that generates this request body is shown below.

```
public static String preregister(String username, String displayName) {
    JsonObjectBuilder payloadBuilder = Json.createObjectBuilder()
        .add(Constants.SKFS_JSON_KEY_USERNAME, username)
        .add(Constants.SKFS_JSON_KEY_DISPLAYNAME, displayName)
        .add(Constants.SKFS_JSON_KEY_OPTIONS, getRegOptions())
        .add("extensions", Constants.JSON_EMPTY);
    return callSKFSRestApi(
        APIURI + Constants.REST_SUFFIX + Constants.PREREGISTER_ENDPOINT,
        payloadBuilder);
}
...
private static String callSKFSRestApi(String requestURI,
JsonObjectBuilder payload){
    JsonObjectBuilder svcinfoBuilder = Json.createObjectBuilder()
        .add("did", SKFSDID)
        .add("protocol", PROTOCOL)
        .add("authtype", Constants.AUTHORIZATION_HMAC);
    JsonObject body = Json.createObjectBuilder()
        .add("svcinfo", svcinfoBuilder)
        .add("payload", payload).build();
...
```

The protocol is used to tell SKFS to generate a challenge for the FIDO2 protocol (instead of the U2F protocol, which SKFS also supports). The *username* and *displayName* are described above. Options are used to specify preferences for the Authenticator used (i.e., a platform vs. cross-platform Authenticator, preferences for Authenticator attestation, preference for user verification vs. user presence, etc.). Extensions are defined in the WebAuthn specification. Options and extensions will not be addressed in this guide.

NOTE: StrongKey's APIs are described in more detail in FIDO2 v3 API Mechanics.

5. In response, SKFS returns a *challenge* to the web application.

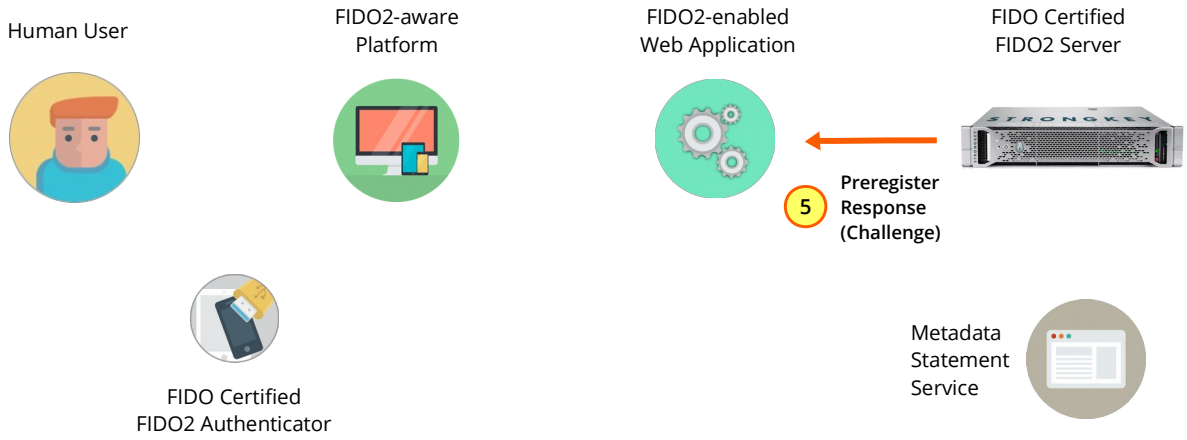


Figure 1—5: Registration: SKFS returns a challenge to the web application.

Upon receiving the *preregister* request, SKFS returns a *challenge* to the service provider web application.

```
{
    "Response": {
        "rp": {
            "name": "StrongKey POC",
            "id": "strongkey.com"
        },
        "user": {
            "name": "johndoe",
            "id": "CXW...FMK4",
            "displayName": "Initial Registration"
        },
        "challenge": "YGmdBIb0JGVE6ZXucUn_Ew",
        "pubKeyCredParams": [{
            "type": "public-key",
            "alg": -7
        }, ...
        {
            "type": "public-key",
            "alg": -39
        }],
        "excludeCredentials": [],
        "attestation": "direct"
    }
}
```

6. The *challenge* passes to the JS code in the web application.

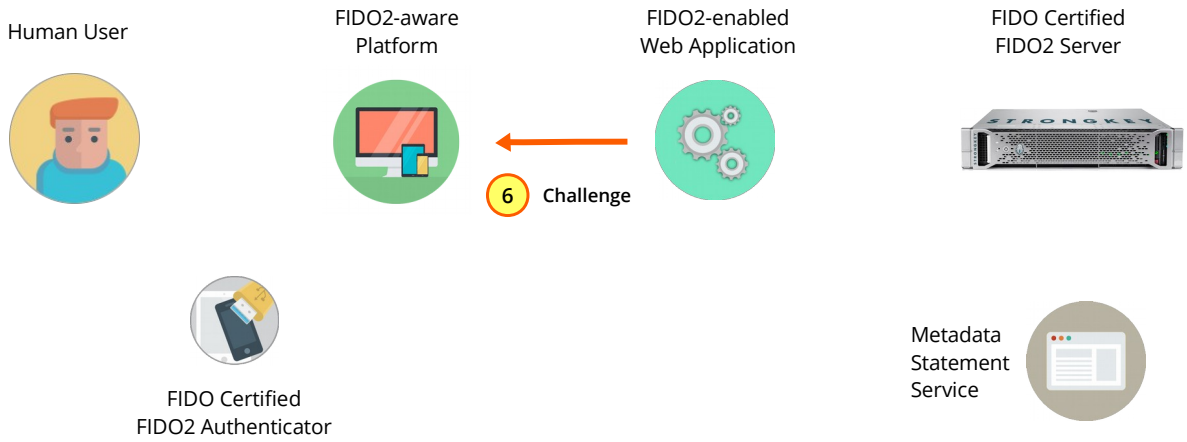


Figure 1—6: Registration: The challenge is passed to the JS front-end code in the web application.

The FIDO2 Server response containing the challenge is base64url-encoded. This must be converted to the `ArrayBuffer` datatype before being sent to the Authenticator.

```
// base64url-decode preregister response from SKFS
let challengeBuffer = this.preregToBuffer(preregResponse);
```

7. The browser code JS sends the *challenge* to the Authenticator.

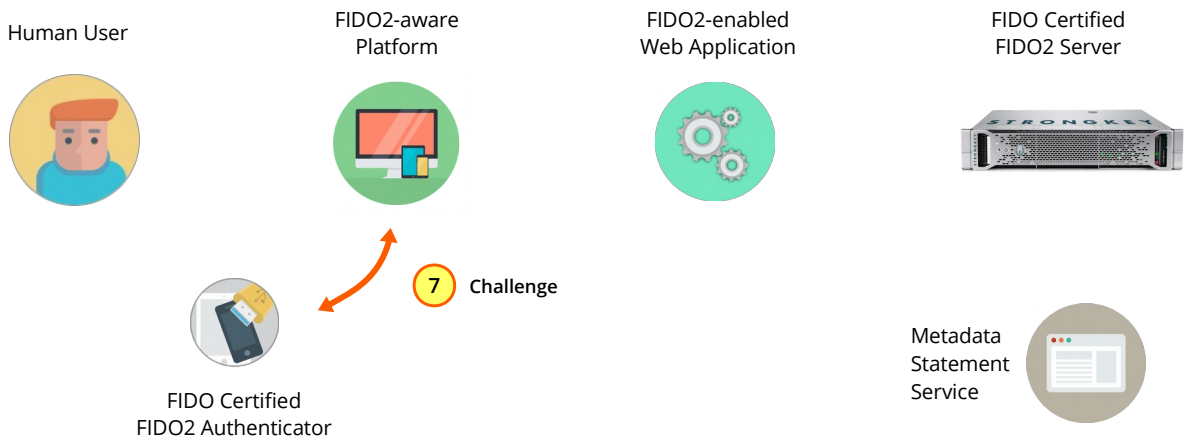


Figure 1—7: Registration: The JS front-end code sends the challenge to the FIDO2 Authenticator.

Having made the JavaScript *preregister* request (Initial Registration, *Step 4*) the JS code receives the *challenge* from the back-end application and delivers the *challenge* to the Authenticator. The JS code is shown here:

```
// Convert base64url fields to ArrayBuffer format [verify].
let challengeBuffer = this.preregToBuffer(preregResponse);
// Browser passes challenge fields to WebAuthn API, which tells relevant
FIDO2 authenticators to generate a new set of public key credentials
let credentialsContainer = window.navigator;
credentialsContainer.credentials.create({ publicKey:
challengeBuffer.Response })
  .then(credResp => {
    // convert response to base64url
    let credResponse = this.preregResponseToBase64(credResp);
      this.post('register', credResponse)
      .done(regResponse => that.onRegResult(regResponse))
        .fail((jqXHR, textStatus, errorThrown) => {
        this.onFailError(jqXHR, textStatus, errorThrown);
        });
      });
```

8. The Authenticator demands a test of human presence.

Figure 1—8: Registration: The Authenticator demands a test of user presence.

Upon receiving the *challenge* and performing internal sanity checks, the Authenticator attached to the user's system will signal the user to confirm user presence by requiring a gesture which may vary by Authenticator—examples might include a button to be pressed, a request for a biometric scan, or a blinking LED. This gesture typically happens by touching the Authenticator.

9. The Authenticator generates a key pair, then digitally signs the response back to the web application JS code.

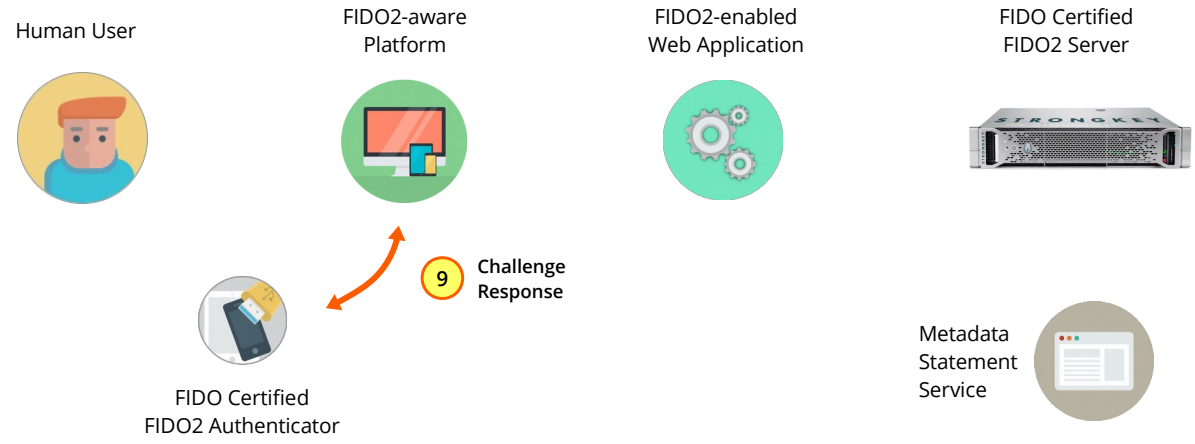


Figure 1—9: Registration: The Authenticator generates a key pair and sends a signed challenge to the web application JavaScript code.

Having received confirmation of the test of user presence, the Authenticator generates a new key pair, creates additional metadata, and sends the response to the JS in the web application front end using one of five different attestation types defined in FIDO2.

10. The JS code sends the response to the web application.

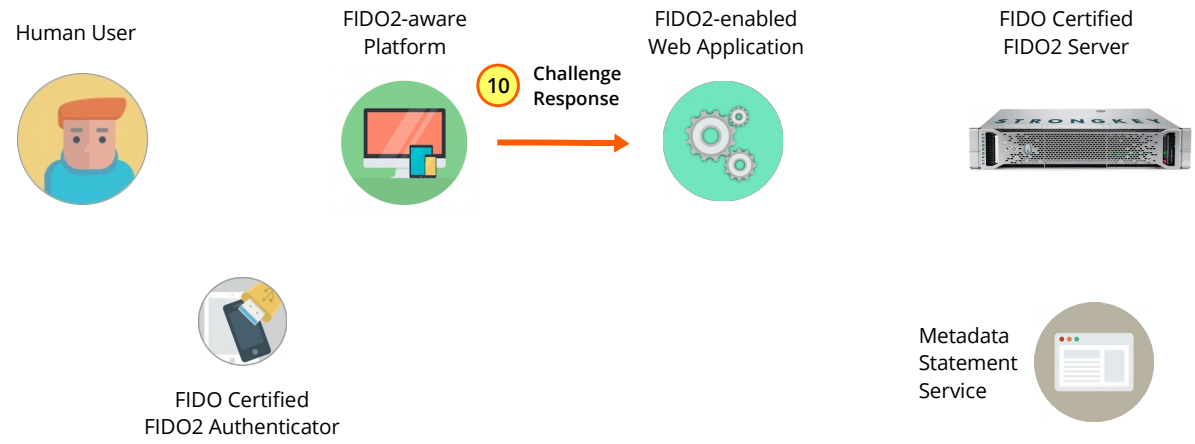


Figure 1—10: Registration: The JS code sends the response to the web application.

To complete the registration process, the JS application now must base64url-encode the response from the Authenticator before sending it to the back end of the service provider web application.

```
// Convert base64url fields to ArrayBuffer format [verify].
let challengeBuffer = this.preregToBuffer(preregResponse);
// Browser passes challenge fields to WebAuthn API, which tells relevant
FIDO2 authenticators to generate a new set of public key credentials
let credentialsContainer = window.navigator;
credentialsContainer.credentials.create({ publicKey:
challengeBuffer.Response })
  .then(credResp => {
    // base64url-encode response from the Authenticator
    let credResponse = this.preregResponseToBase64(credResp);
      // Call the register web service on the web app back end
      this.post('register', credResponse)
      .done(regResponse => that.onRegResult(regResponse))
        .fail((jqXHR, textStatus, errorThrown) => {
        this.onFailError(jqXHR, textStatus, errorThrown);
        });
      });
```

11. The web application calls the *register* web service on SKFS.

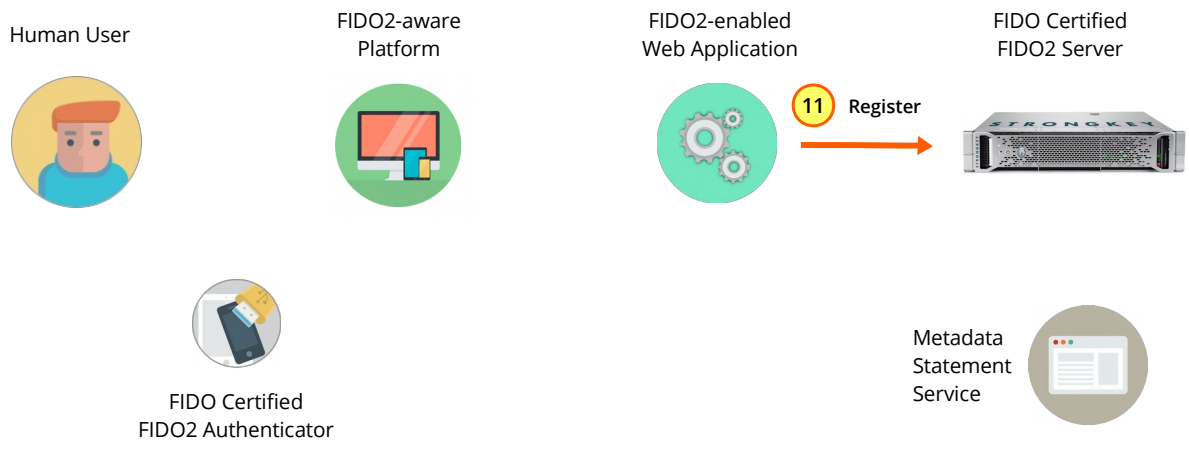


Figure 1—11: Registration: The web application back end calls the register web service on SKFS.

The SKFS *register* web service must receive this JSON data structure (which is not displayed in full for brevity's sake); for the full response, see FIDO2 v3 API registration Request Body):

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "publicKeyCredential": {
      "id": "MBDVx...c8wA",
      "rawId": "MBDVx...c8wA",
      "response": {
          "attestationObject": "o2Nm...ZqFA",
          "clientDataJSON": "eyJ0...bSJ9"
      },
      "type": "public-key"
    },
    "strongkeyMetadata": {
      "version": "1.0",
      "create_location": "Sunnyvale, CA",
      "origin": "https://<FQDN>",
      "username": "johndoe"
    }
  }
}
```

The Authenticator response is sent to SKFS via a *register* web service request. The Java code that generates this request is shown below.

```
public static String register(String username, String origin, JsonObject
signedResponse) {
    JsonObject reg_metadata = javax.json.Json.createObjectBuilder()
        .add("version", PROTOCOL_VERSION) // ALWAYS since this is just
the first revision of the code
        .add("create_location", "Sunnyvale, CA")
        .add("username", username)
        .add("origin", origin).build();
    JsonObjectBuilder reg_inner_response =
javax.json.Json.createObjectBuilder()
        .add("attestationObject",
signedResponse.getJsonObject("response").getString("attestationObject"))
        .add("clientDataJSON",
signedResponse.getJsonObject("response").getString("clientDataJSON"));
    JsonObject reg_response = javax.json.Json.createObjectBuilder()
        .add("id", signedResponse.getString("id"))
```

```
        .add("rawId", signedResponse.getString("rawId"))
        .add("response", reg_inner_response) // inner response object
        .add("type", signedResponse.getString("type")).build();
    JsonObjectBuilder payloadBuilder = Json.createObjectBuilder()
        .add("publicKeyCredential", reg_response)
        .add("strongkeyMetadata", reg_metadata);
    return callSKFSRestApi(
        APIURI + Constants.REST_SUFFIX + Constants.REGISTER_ENDPOINT,
        payloadBuilder);
}
...
private static String callSKFSRestApi(String requestURI,
JsonObjectBuilder payload){
    JsonObjectBuilder svcinfoBuilder = Json.createObjectBuilder()
        .add("did", SKFSDID)
        .add("protocol", PROTOCOL)
        .add("authtype", Constants.AUTHORIZATION_HMAC);
    JsonObject body = Json.createObjectBuilder()
        .add("svcinfo", svcinfoBuilder)
        .add("payload", payload).build();
...
```

12. SKFS verifies response and optionally validates the Authenticator's attestation.

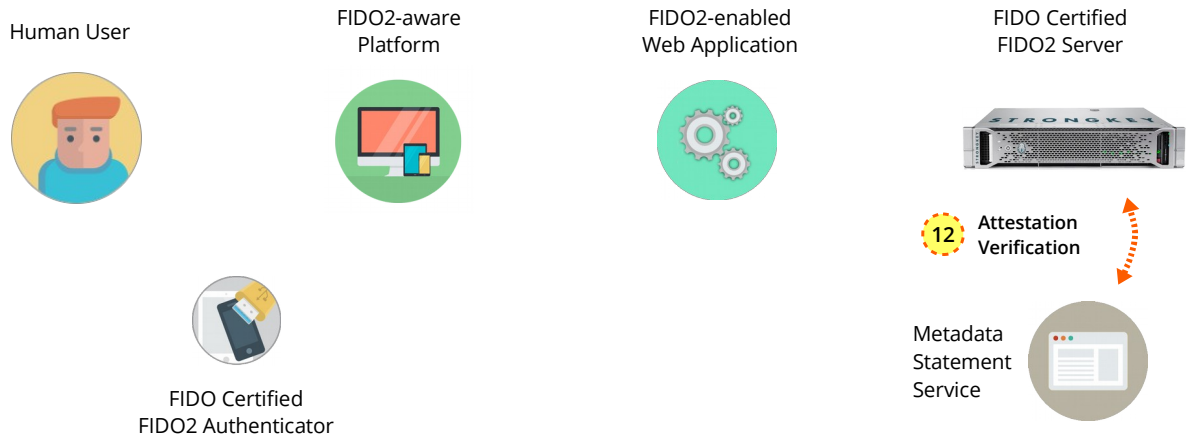


Figure 1–12: Registration: SKFS validates the challenge response and (optionally) attestation.

13. SKFS registers the user's public key by storing it.

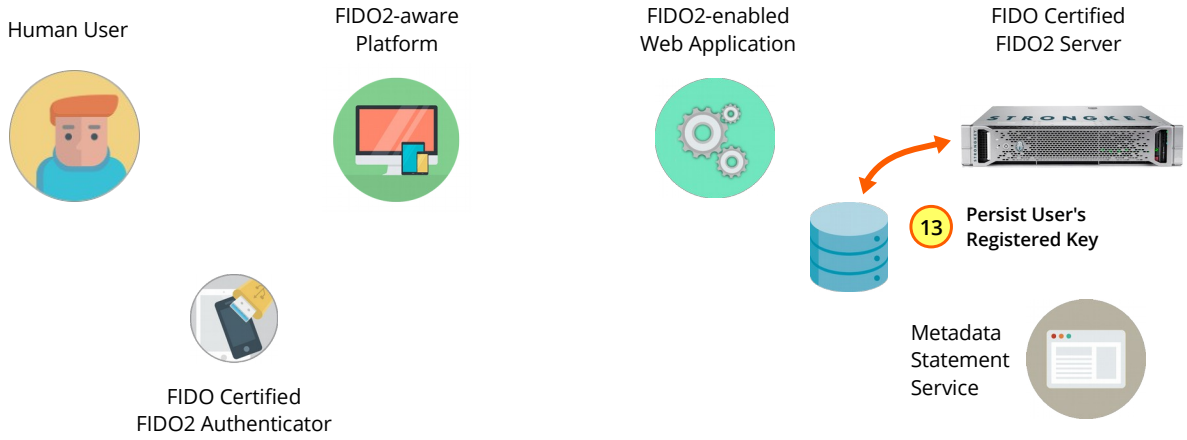


Figure 1—13: Registration: The user's public key is persisted into the database.

On successful verification of the Authenticator's response, StrongKey's FIDO2 Server will store the user's public key in its database.

14. The success response is sent to the service provider web application.

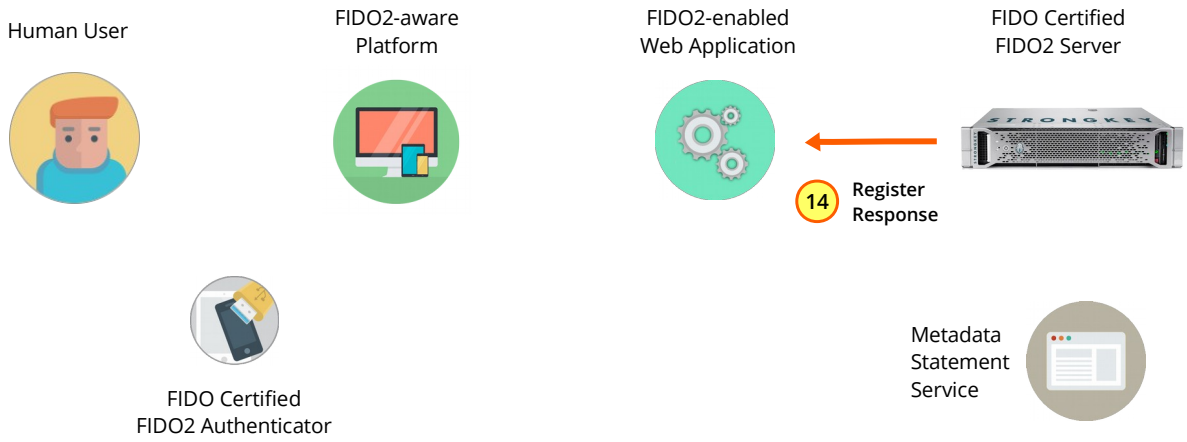


Figure 1—14: Registration: The success response is returned to the web application.

Upon receiving the *register* request and processing the Authenticator's response, SKFS returns an appropriate response to the service provider web application.

15. FIDO2 registration is stored and enabled for this user.

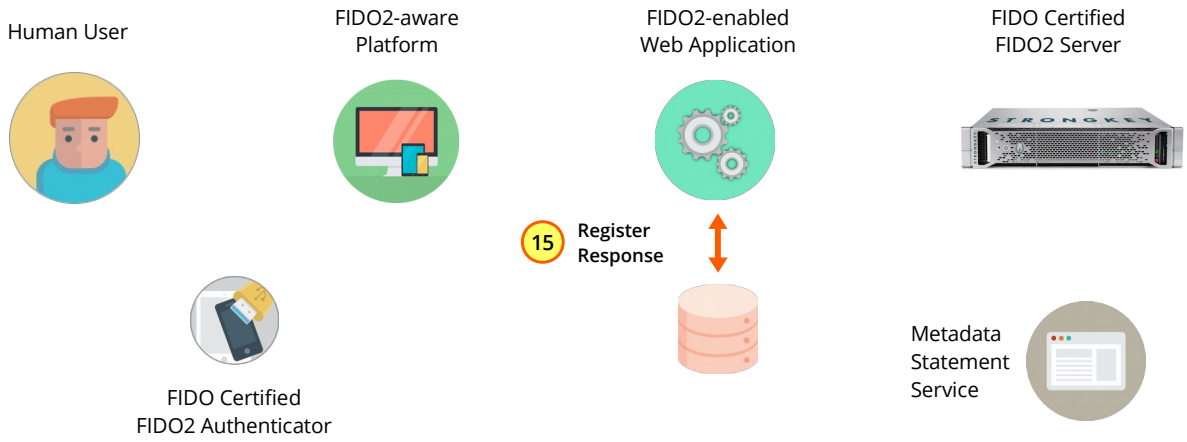


Figure 1—15: Registration: FIDO2 is enabled for the user in the web application database.

Optionally, the service provider web application may store information about the successful registration in its own database and perform additional processing as needed.

16. The success response is sent to the web application JS code.

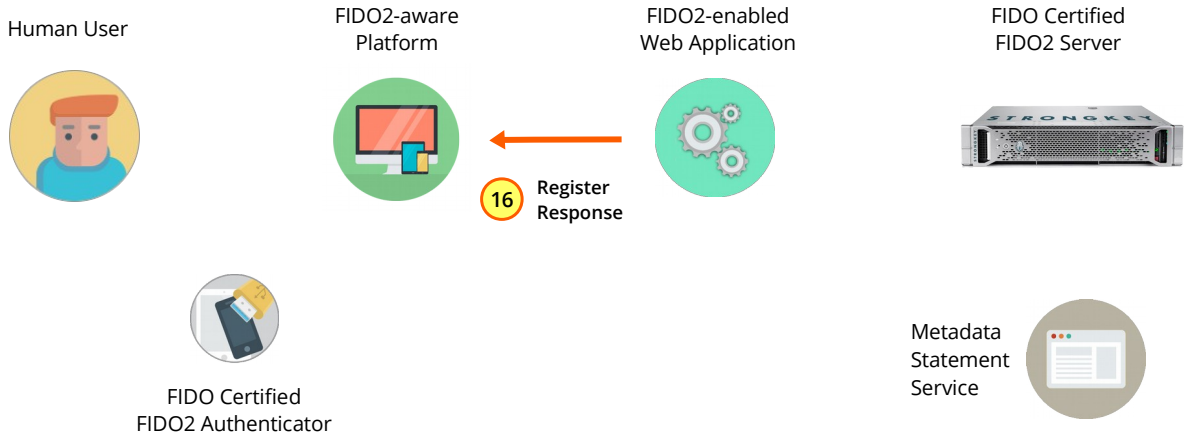


Figure 1—16: Registration: The success response is sent to the browser.

The message received (typically an *HTTP 200 OK*) by the web application front end is in response to its own *register* web service.

17. The user is notified that they are now registered with a FIDO2 key.

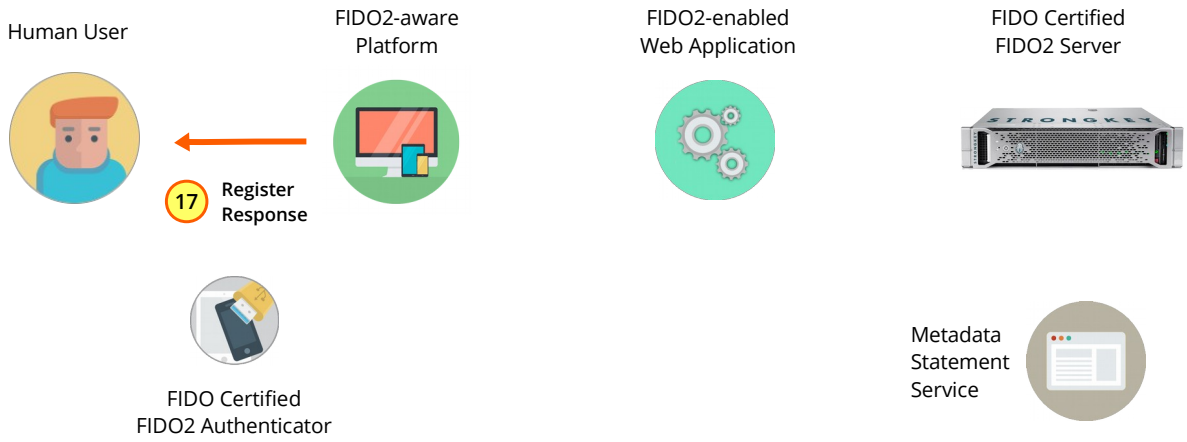


Figure 1—17: Registration: Success! The user has completed initial registration.

4.1.2—Authentication

In a typical password-based authentication, a web application provides a form asking for the username used during registration and a password. Upon receiving the username and password, the web application typically passes the username and password to its authentication module. The authentication module then verifies that salting and hashing the password received for that username matches what is in the database. If it matches, it will return a success value and login the user.

In a FIDO2 authentication workflow, the web application sends a *challenge* to the user to sign using their Authenticator. The signed *challenge* is then passed back to the service provider application, which relays the signature to the FIDO2 server to verify that the challenge was properly signed using the public key stored during registration. The steps to achieve this are listed below.

1. The user completes a login form and supplies their *username*.

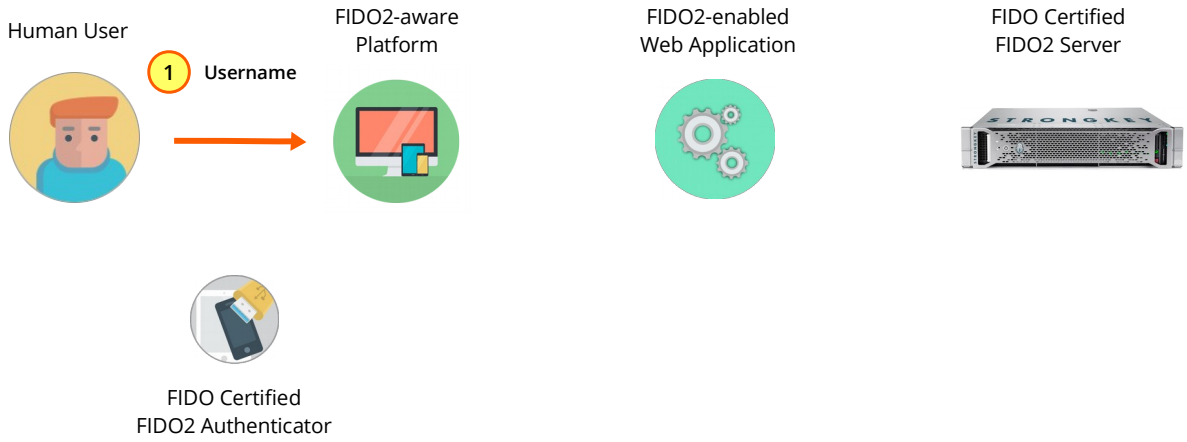


Figure 1—18: Authentication: The user supplies the username to a login form.

During FIDO2 authentication, the web application typically provides a form asking for the username; the HTML is shown here:

```
<div id="authPanel" class="form-panel" style="display: none;">
  <div id="authHeadingPanel">
    <img width="24" src="img/login256.png">
    <h2>Sign In</h2>
  </div>
  <div>
    <input class="met" id="authUsername" placeholder="Username"
required="" type="text">
  </div>
  <div>
    <button class="met" id="authSubmit">Sign In</button>
  </div>
</div>
```

2. The user submits *username* information to the web application.

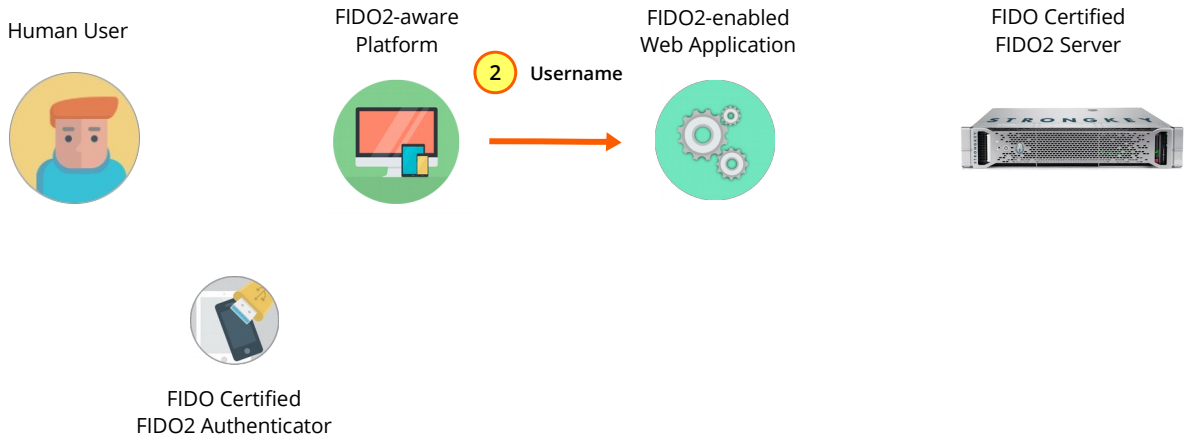


Figure 1–19: Authentication: User submits login credentials to the web application.

Upon the user submitting the information to the web form, the service provider's web application JS front end validates the information and calls a web service—named 'preauthenticate' in the JS code to align with the SKFS *preauthenticate* web service—to handle the authentication process.

```
this.post('preauthenticate', {
  'username': $('#authUsername').val()
})
.done((resp) => {
  this.authenticate(resp.Response);
})
.fail((jqXHR, textStatus, errorThrown) => {
  this.onFailError(jqXHR, textStatus, errorThrown);
});
```

3. The user's *username* and FIDO2-enablement status is verified.

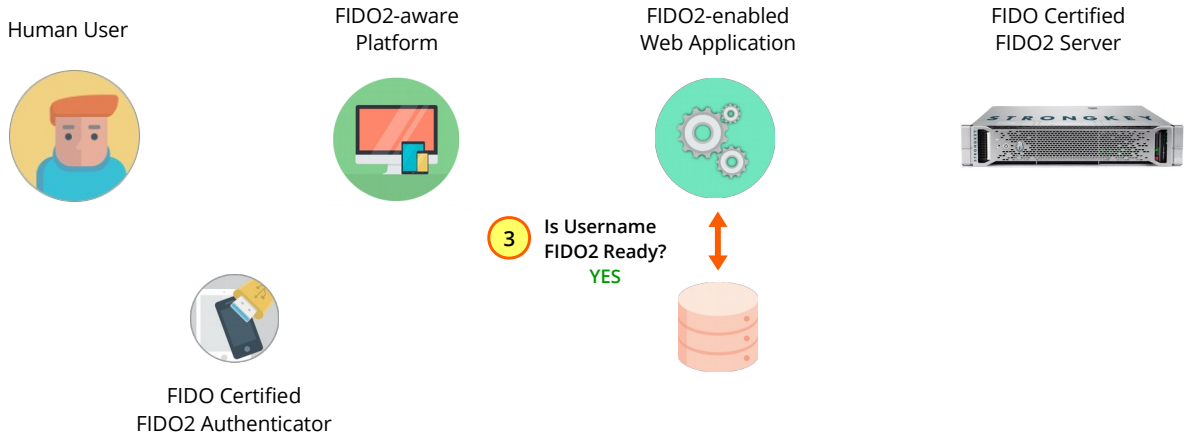


Figure 1—20: Authentication: The user's username is verified for FIDO2-enablement.

Upon receiving the *username*, the service provider's web application back end verifies the *username* has previously registered a FIDO2 key with this site. If either verification fails to pass the test, the back-end application must respond appropriately to the user. If the *username* is already in use and if the user has registered a FIDO2 key, the web application proceeds.

```
// Get user input + basic input checking
  String username = getValueFromInput(Constants.RP_JSON_KEY_USERNAME,
input);
  // Verify user exists
  if(!userdatabase.doesUserExist(username)){
    WebauthnTutorialLogger.logp(Level.SEVERE, CLASSNAME,
"preauthenticate", "WEBAUTHN-WS-ERR-1002", username);
    return generateResponse(Response.Status.NOT_FOUND,
WebauthnTutorialLogger.getMessageProperty("WEBAUTHN-WS-ERR-1002"));
  }
```

4. The web application sends a *preauthenticate* web service request to SKFS.

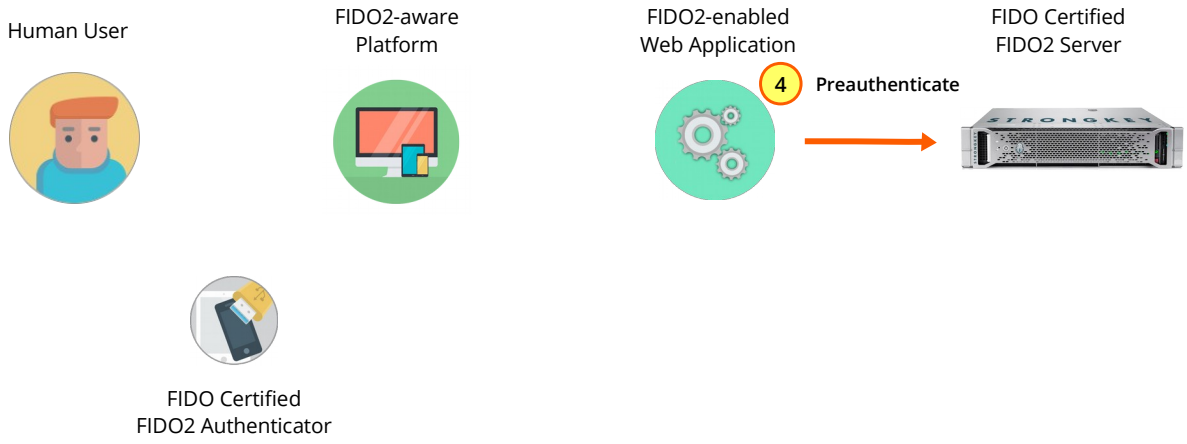


Figure 1—21: Authentication: A preauthenticate web service request is sent to SKFS.

The JSON data structure required for the SKFS *preauthenticate* web service follows:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "user.name@domain.com",
    "options": {}
  }
}
```

The service provider's web application assembles the JSON data structure first and then sends a *preauthenticate* web service request to SKFS using the REST protocol. The Java code that performs this process is shown below.

```
String preauth = SKFSClient.preauthenticate(username);
HttpSession session = request.getSession(true);
session.setAttribute(Constants.SESSION_USERNAME, username);
session.setAttribute(Constants.SESSION_ISAUTHENTICATED, false);
return generateResponse(Response.Status.OK, preauth);

public static String preauthenticate(String username) {
    JsonObjectBuilder payloadBuilder = Json.createObjectBuilder()
        .add(Constants.SKFS_JSON_KEY_USERNAME, username)
        .add(Constants.SKFS_JSON_KEY_OPTIONS, getAuthOptions());
    return callSKFSRestApi(
        APIURI + Constants.REST_SUFFIX +
```

```
Constants.PREAUTHENTICATE_ENDPOINT,
        payloadBuilder);
}
...
private static String callSKFSRestApi(String requestURI,
JsonObjectBuilder payload){
    JsonObjectBuilder svcinfoBuilder = Json.createObjectBuilder()
        .add("did", SKFSDID)
        .add("protocol", PROTOCOL)
        .add("authtype", Constants.AUTHORIZATION_HMAC);
    JsonObject body = Json.createObjectBuilder()
        .add("svcinfo", svcinfoBuilder)
        .add("payload", payload).build();
...
```

5. In response, SKFS returns a *challenge* to the web application.

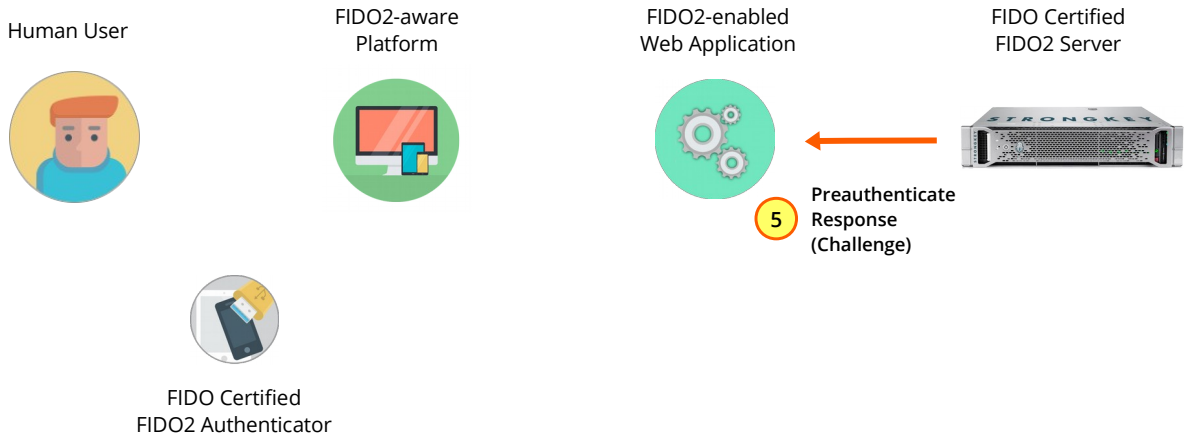


Figure 1—22: Authentication: SKFS returns a challenge to the service provider web application.

Upon receiving a *preauthenticate* request, StrongKey's FIDO2 Server confirms the user exists with a registered public key and returns a *challenge* to the web application. The abbreviated SKFS response is shown here (for the full response see the FIDO2 v3 API preauthenticate Example):

```
{
    "Response": {
        "challenge": "k1YeYZZ6HDmg3ruKinb2SQ",
        "allowCredentials": [{
            "type": "public-key",
            "id": "WLwV...CuqQ",
            "alg": -7
        }],
        "rpId": "strongkey.com"
    }
}
```

6. The web application back end passes the *challenge* to the JS code in the front end.

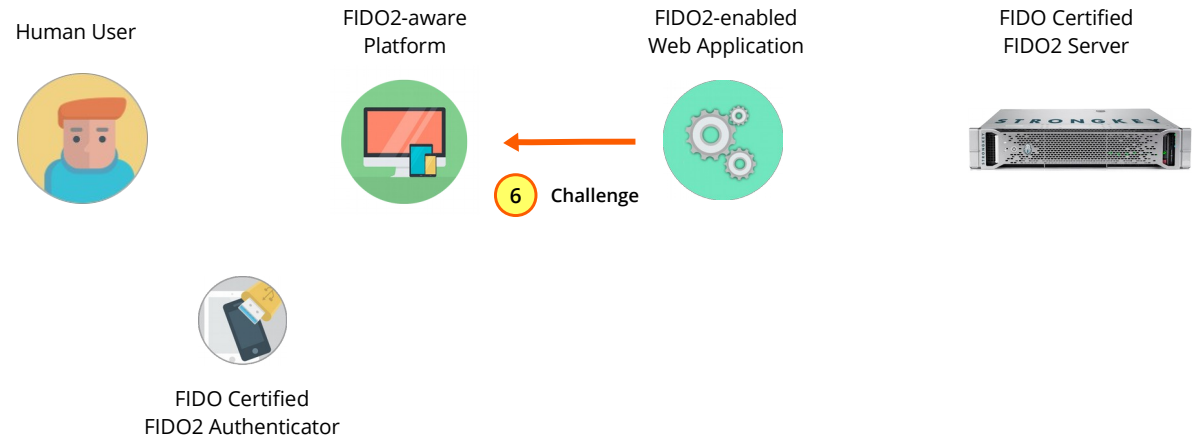


Figure 1–23: Authentication: The web application passes the challenge to the front end.

The FIDO2 Server response containing the Authentication *challenge* is base64url-decoded. This must be converted from the `ArrayBuffer` datatype before being sent to the Authenticator:

```
// base64url-decode fields to ArrayBuffer format.
let challengeBuffer = this.preauthToBuffer(preauthResponse);
```

7. The JavaScript code passes the *challenge* to the Authenticator.

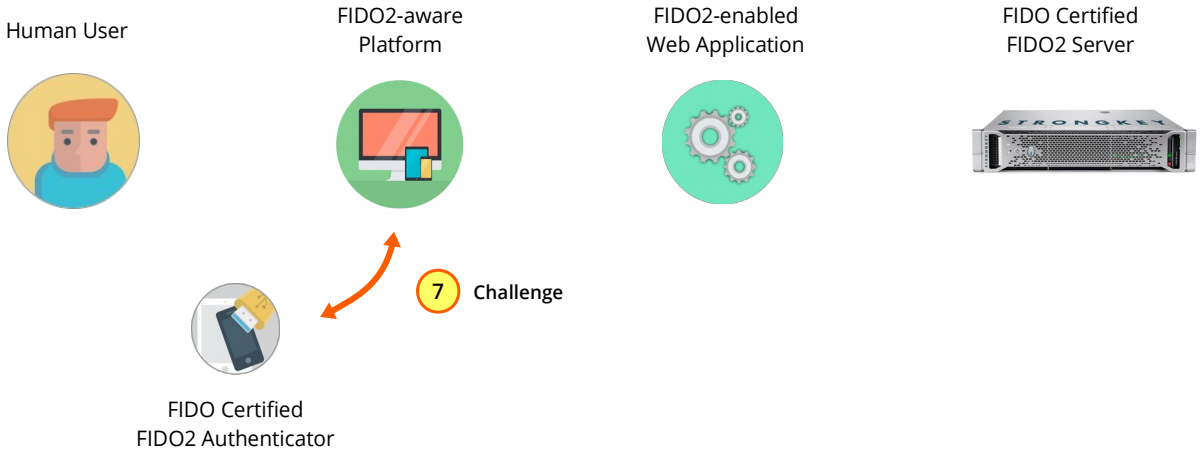


Figure 1–24: Authentication: The Authenticator receives the challenge from the JS front end.

```
// base64url-decode fields to ArrayBuffer format.
  let challengeBuffer = this.preauthToBuffer(preauthResponse);

// The browser passes the challenge to the Authenticator, which requests
a signature.
  let credentialsContainer;
  credentialsContainer = window.navigator;
  credentialsContainer.credentials.get({ publicKey:
challengeBuffer.Response })
      .then(credResp => {
  // base64url-encode the response from the Authenticator
  let credResponse = that.preauthResponseToBase64(credResp);
  this.post('authenticate', credResponse)
      .done(authResponse => that.onAuthResult(authResponse))
      .fail((jqXHR, textStatus, errorThrown) => {
  this.onFailError(jqXHR, textStatus, errorThrown);
      });
      })
```

8. Origin verification and test of user presence confirmation.

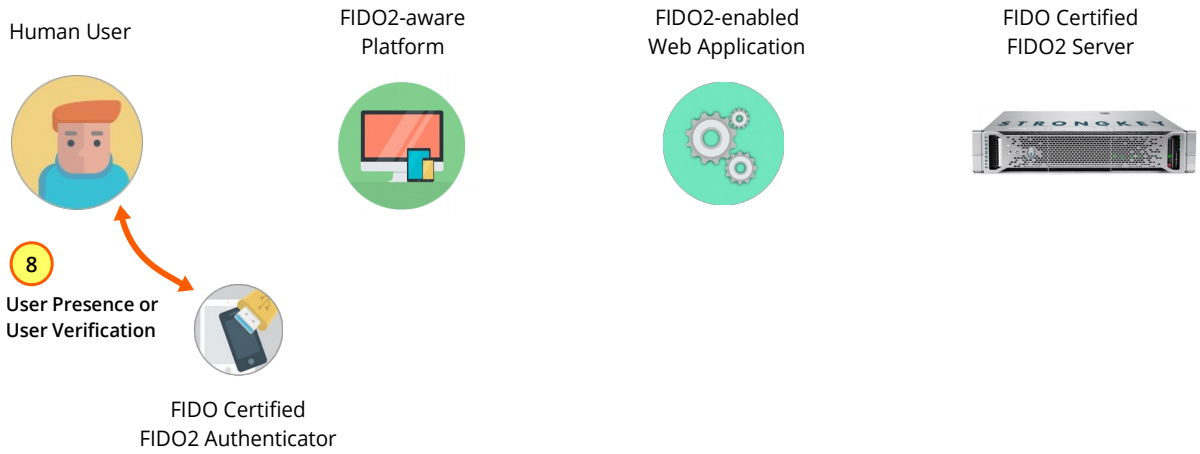


Figure 1—25: Authentication: Test of user presence confirmation from the user.

Upon receiving the *challenge* and performing internal sanity checks the Authenticator attached to the user's system will signal the user to confirm user presence by requiring a gesture which may vary by Authenticator—examples might include a button to be pressed, a request for a biometric scan, or a blinking LED. This gesture typically happens by touching the Authenticator.

NOTE: The Authenticator verifies the web application's site origin matches the credential's origin against which the key was registered by this user. This is a unique security feature of the FIDO2 protocol to ensure phishing websites cannot activate users' credentials.

9. Authenticator signs the *challenge* with user's private key.

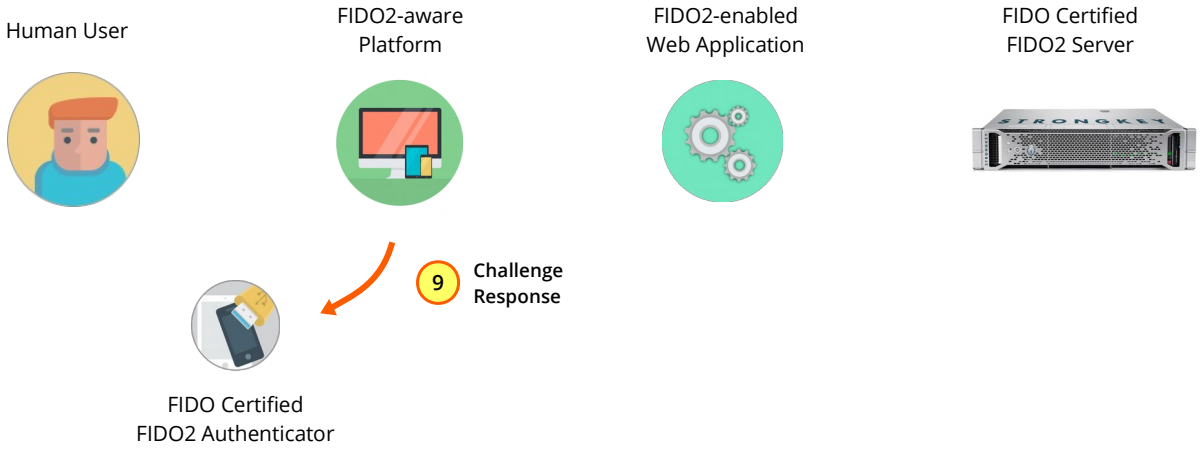


Figure 1–26: Authentication: The Authenticator-signed challenge is sent to the JS front end.

Having received confirmation of the test of user presence, the Authenticator uses the existing credential's private key, creates additional metadata, and sends the response to the JS in the web application front end.

10. The signed *challenge* is sent to the service provider's web application back end.

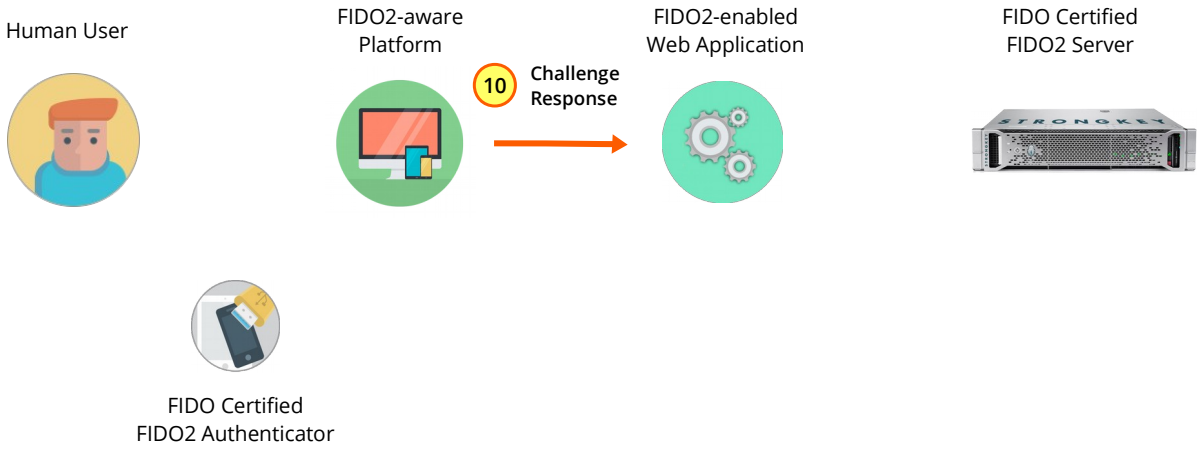


Figure 1–27: Authentication: The signed challenge is sent to the web application back end.

To complete the authentication process, the JS application now must base64url-encode the response from the Authenticator before sending it to the back end of the service provider web application.

```
// base64url-decode fields to ArrayBuffer format.
  let challengeBuffer = this.preauthToBuffer(preauthResponse);
```

```
// The browser passes the challenge to the Authenticator, which requests
a signature.
  let credentialsContainer;
  credentialsContainer = window.navigator;
  credentialsContainer.credentials.get({ publicKey:
challengeBuffer.Response })
      .then(credResp => {
  // base64url-encode the response from the Authenticator
  let credResponse = that.preauthResponseToBase64(credResp);
  this.post('authenticate', credResponse)
      .done(authResponse => that.onAuthResult(authResponse))
      .fail((jqXHR, textStatus, errorThrown) => {
  this.onFailError(jqXHR, textStatus, errorThrown);
      });
      })
```

11. The web application calls the *authenticate* web service on the SKFS.

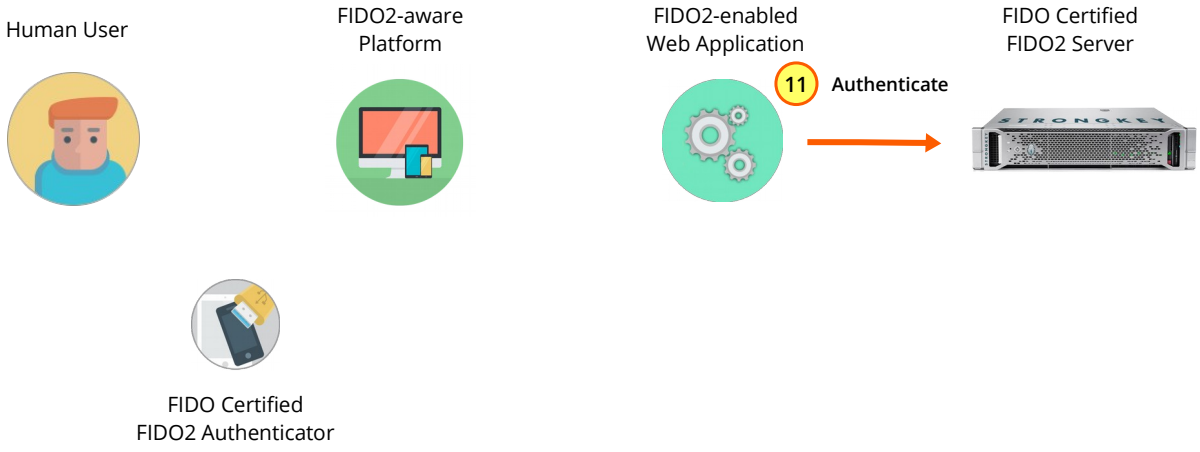


Figure 1—28: Authentication: Web application calls the authenticate web service on the SKFS.

The SKFS *authenticate* web service must receive this JSON data structure (which is not displayed in full for brevity's sake); for the full response, see 4.2.5.1—FIDO2 v3 API authenticate Request Body):

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "publicKeyCredential": {
      "id": "WLwV...CuqQ",
```

```
      "rawId": "WLwV...CuqQ",
      "response": {
        "authenticatorData": "UoqJ...ABZA",
        "signature": "MEUC...r124",
        "userHandle": "",
        "clientDataJSON": "eyJj...dCJ9"
      },
      "type": "public-key"
    },
    "strongkeyMetadata": {
      "version": "1.0",
      "last_used_location": "Cupertino, CA",
      "username": "johndoe",
      "origin": "https://<FQDN>"
    }
  }
}
```

The Authenticator response is sent to SKFS via an *authenticate* web service. The Java code that generates this request is shown below.

```
public static String authenticate(String username, String origin,
JsonObject signedResponse) {
    JsonObject auth_metadata = javax.json.Json.createObjectBuilder()
        .add("version", PROTOCOL_VERSION) // ALWAYS since this is just
the first revision of the code
        .add("last_used_location", "Sunnyvale, CA")
        .add("username", username)
        .add("origin", origin)
        .build();
    JsonObjectBuilder auth_inner_response =
javax.json.Json.createObjectBuilder()
        .add("authenticatorData",
signedResponse.getJsonObject("response").getString("authenticatorData"))
        .add("signature",
signedResponse.getJsonObject("response").getString("signature"))
        .add("userHandle",
signedResponse.getJsonObject("response").getString("userHandle"))
        .add("clientDataJSON",
signedResponse.getJsonObject("response").getString("clientDataJSON"));
    JsonObject auth_response = javax.json.Json.createObjectBuilder()
        .add("id", signedResponse.getString("id"))
        .add("rawId", signedResponse.getString("rawId"))
        .add("response", auth_inner_response) // inner response object
        .add("type", signedResponse.getString("type"))
        .build();
    JsonObjectBuilder payloadBuilder = Json.createObjectBuilder()
        .add("publicKeyCredential", auth_response)
        .add("strongkeyMetadata", auth_metadata);
```

```
    return callSKFSRestApi(
        APIURI + Constants.REST_SUFFIX + Constants.AUTHENTICATE_ENDPOINT,
        payloadBuilder);
}
...
private static String callSKFSRestApi(String requestURI,
JsonObjectBuilder payload){
    JsonObjectBuilder svcinfoBuilder = Json.createObjectBuilder()
        .add("did", SKFSDID)
        .add("protocol", PROTOCOL)
        .add("authtype", Constants.AUTHORIZATION_HMAC);
    JsonObject body = Json.createObjectBuilder()
        .add("svcinfo", svcinfoBuilder)
        .add("payload", payload).build();
...
```

12. SKFS verifies the *challenge* response and updates relevant metadata.

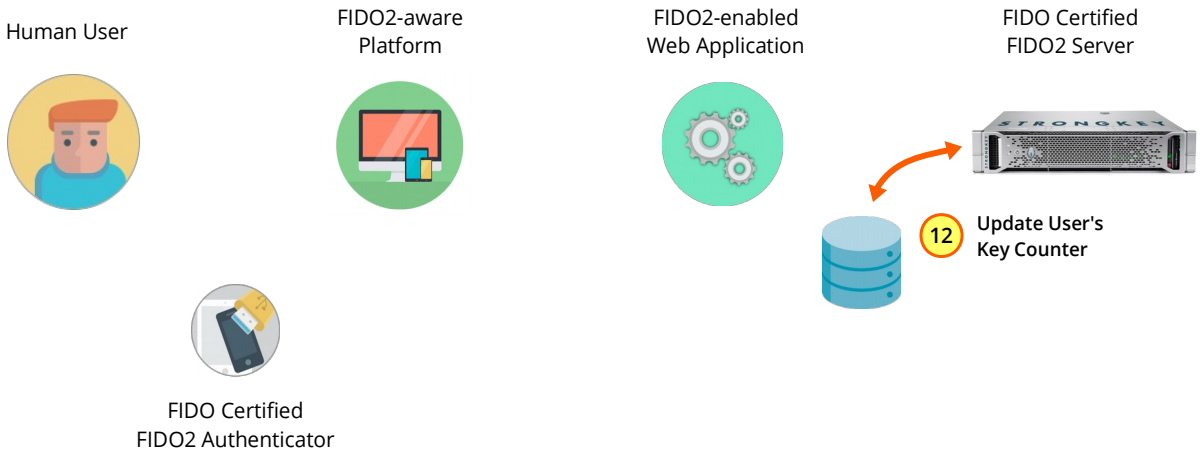


Figure 1–29: Authentication: SKFS verifies the challenge response and updates the key counter.

SKFS verifies the *challenge* using the public key that was stored for the user during registration.

13. SKFS returns a success message to the web application.

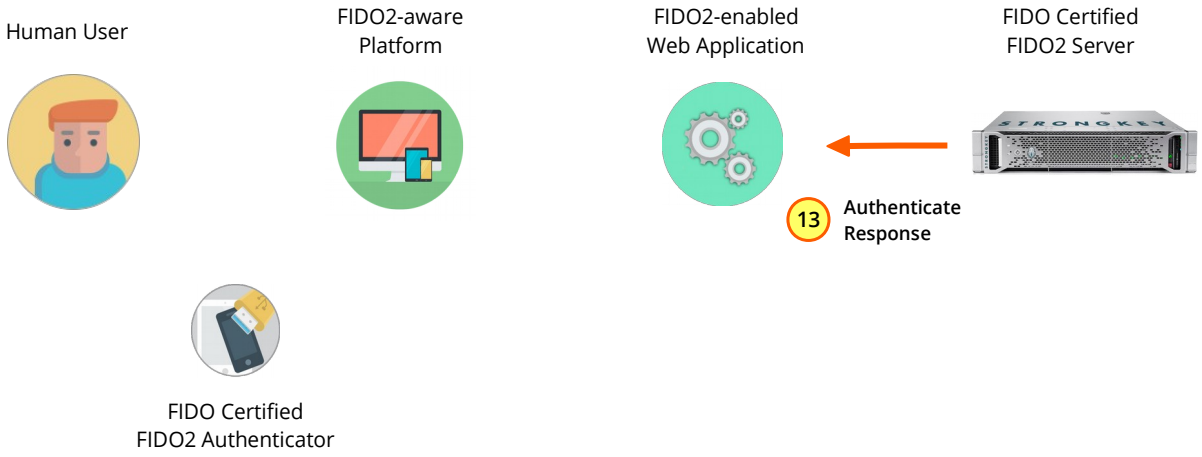


Figure 1—30: Authentication: SKFS returns a success response.

If verification succeeds, the FIDO2 Server returns a success message to the service provider web application .

14. The service provider web application performs post-success housekeeping.

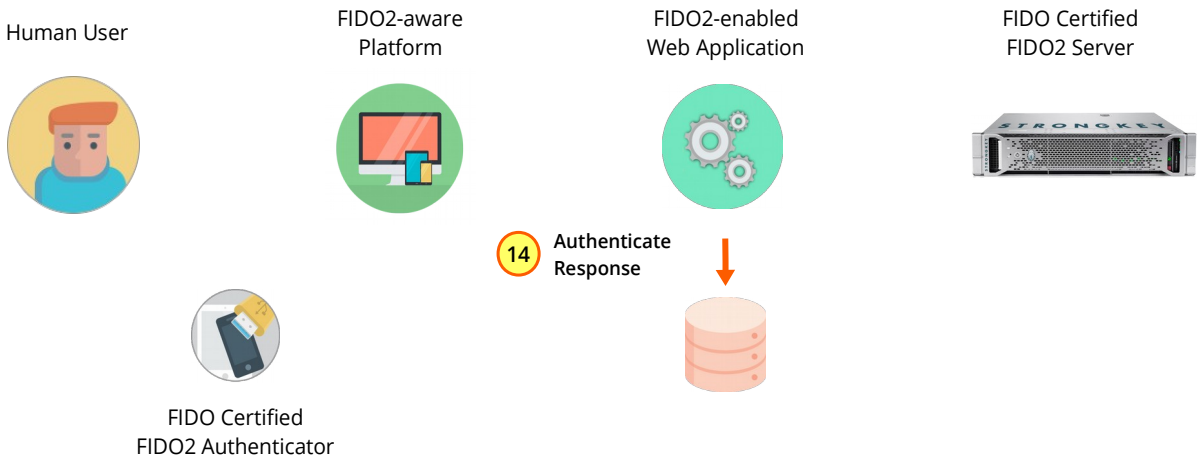


Figure 1—31: Authentication: The service provider web application stores the response.

Upon successfully FIDO2-authenticating a user, the web application may perform additional processing tasks; for example:

- establishing a sessionID
- determining the user's authorization(s) within the applications
- updating application-level metadata

15. The service provider web application returns the success response to the JS front end.

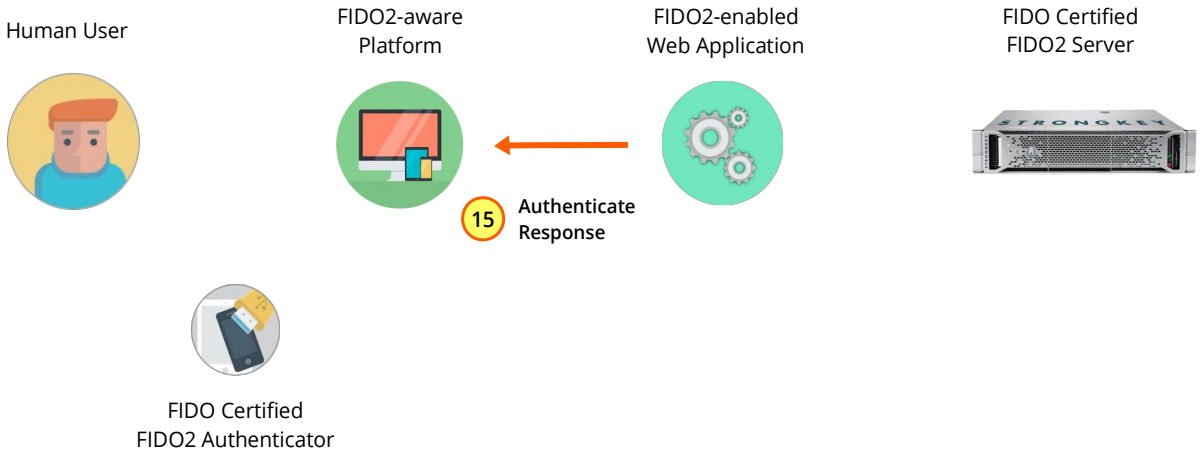


Figure 1—32: Authentication: The web application sends the response to the JavaScript front end.

The message received (typically an *HTTP 200 OK*) by the web application front end is in response to its own *authenticate* web service.

16. Success! The user is notified of a successful FIDO2 login.

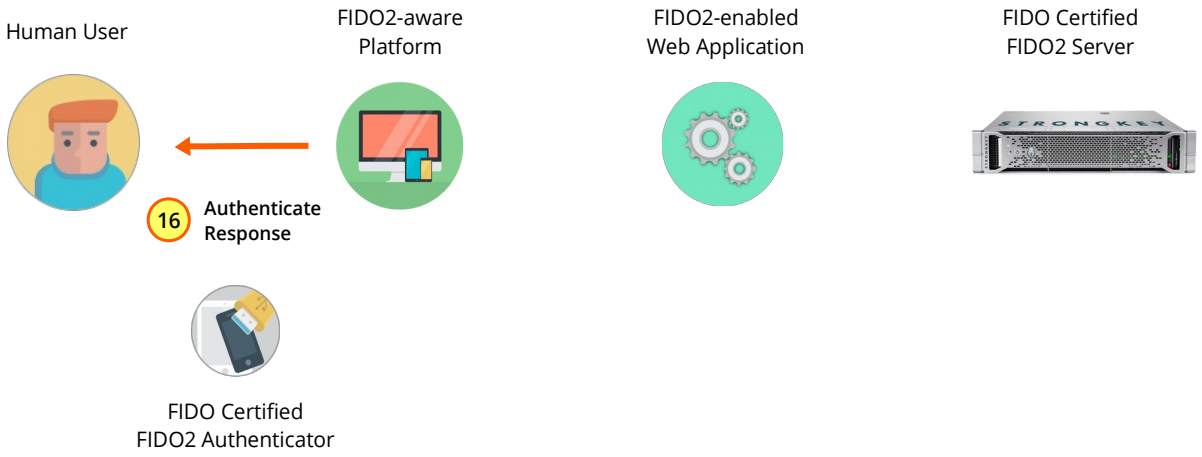


Figure 1—33: Authentication: FIDO2 login is successful!

4.2—FIDO2 v3 API Mechanics

The FIDO2 v3 web service application supports an array of web services for registering, authenticating, and managing FIDO keys. It works by having client applications send requests to the FIDO2 v3 API through a standard *Representational State Transfer (REST)*-based web services over the *Secure Hyper Text Transfer Protocol (HTTPS)*.

REST-based web service calls will always return a *200 OK* alongside a response.

4.2.1—FIDO2 v3 API Web Service Description

The web service description for REST can be found at the following URL on your Tellaro:

- /api/application.wadl

4.2.2—FIDO v3 API REST-based *preregister* Mechanics

The *preregister* operation is used to generate a challenge to be signed by the FIDO Authenticator. The web service receives a set of parameters that contains all the metadata necessary to create a challenge.

Parameter	Explanation
svcinfo	Object that carries FIDO2 v3 API service information.
payload	Object that carries the *preregister* web service parameters. See below for the *FIDO2_0* request body.

4.2.2.1—FIDO2 v3 API *preregister* Request Body

The following request body is sent during the *preregister* call:

- URL: https://<FQDN>:<PORT>/skfs/rest/preregister
- HTTP Method: POST
- *FIDO2_0* request body:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe",
    "displayName": "Initial Registration",
    "options": {
      "attestation": "direct"
    },
    "extensions": "{}"
  }
}
```

Value	Explanation
did	Unique domain identifier for a cryptographic domain.
protocol	The FIDO protocol to be used in this request (FIDO2_0).
authtype	Type of authorization used to request the service. Can be *PASSWORD* or *HMAC*.
svcusername	Username requesting the service. The service credentials are looked up in the 'service' setup of authentication system based on *Lightweight Directory Access Protocol (LDAP)/Active Directory (AD)*.
svcpassword	Password of the service username specified above.
username	The user account to which to register the FIDO key.
displayName	A user-friendly name assigned as a label to this key (e.g., “Blue Yubikey”).
options	A JSON containing a list of options to associate with this challenge. The options JSON can be empty, but is otherwise a required parameter.
extensions	A JSON containing a list of extensions to associate with this challenge. The extensions JSON can be empty, but is otherwise a required parameter. More details.

4.2.2.2—FIDO2 v3 API *preregister* Example

- Successful *FIDO2_0* response (accompanied by a *200 OK* if using REST):

```
{
  "Response": {
        "rp": {
              "name": "StrongKey POC",
              "id": "strongkey.com"
        },
        "user": {
              "name": "johndoe",
              "id": "CXWhkBcUyErINZ_FDSuOVVdmu0YzT0HkVSONouBFMK4",
              "displayName": "Initial Registration"
        },
        "challenge": "YGmdBIb0JGVE6ZXucUn_Ew",
        "pubKeyCredParams": [{
              "type": "public-key",
              "alg": -7
        }, {
              "type": "public-key",
              "alg": -35
        }, {
              "type": "public-key",
              "alg": -36
        }, {
              "type": "public-key",
              "alg": -8
        }, {
              "type": "public-key",
              "alg": -43
        }, {
```

```
            "type": "public-key",
            "alg": -65535
        }, {
            "type": "public-key",
            "alg": -257
        }, {
            "type": "public-key",
            "alg": -258
        }, {
            "type": "public-key",
            "alg": -259
        }, {
            "type": "public-key",
            "alg": -37
        }, {
            "type": "public-key",
            "alg": -38
        }, {
            "type": "public-key",
            "alg": -39
        }],
        "excludeCredentials": [],
        "attestation": "direct"
    }
}
```

4.2.3—FIDO v3 API *register* Mechanics

The *register* operation immediately follows a call to *preregister* and is used to complete registration of a FIDO Authenticator. The web service receives a set of parameters that contains all the metadata necessary to register the key.

Parameter	Explanation
svcinfo	Object that carries FIDO2 v3 API service information.
payload	Object that carries the *register* web service parameters. See below for the *FIDO2_0* request body.

When the Tellaro receives the request, it verifies the credentials presented against its internal directory server, and determines their authorization to request the *register* service by verifying if they are a member of the *FidoAuthorized* groups.

4.2.3.1—FIDO2 v3 API *registration* Request Body

The following request body is sent during the registration call:

- URL: https://<FQDN>:<PORT>/skfs/rest/register
- HTTP Method: POST

- *FIDO2_0* request body:


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "publicKeyCredential": {
      "id":
"MBDVxPOZ5To939FLGuhTPaaMA1jqTvajZrqWKbnI81yhEndkjQPbL7Q6W5TerIq_rowNstdv
rXCLsOw4aO1-xJB-Q4-
WkNPMdYhIiN9yt0rRIiev917ezeNzwIosjrN99MUHR_J_Sw6Js4Q49mllAgZ-
gaxnqd7pmIX_V6B7oDfWaKmvImwxo3pGXqXb-6pboouYVbiMl6WA-
TooklND0pIXWxdp2SvbfkoIur-c8wA",
      "rawId":
"MBDVxPOZ5To939FLGuhTPaaMA1jqTvajZrqWKbnI81yhEndkjQPbL7Q6W5TerIq_rowNstdv
rXCLsOw4aO1-xJB-Q4-
WkNPMdYhIiN9yt0rRIiev917ezeNzwIosjrN99MUHR_J_Sw6Js4Q49mllAgZ-
gaxnqd7pmIX_V6B7oDfWaKmvImwxo3pGXqXb-6pboouYVbiMl6WA-
TooklND0pIXWxdp2SvbfkoIur-c8wA",
      "response": {
        "attestationObject":
"o2NmbXRmcGFja2VkZ2F0dFN0bXSjY2FsZyZjc2lnWEgwRgIhAJ4iYNBFTz_LTi37Dts5HDpH
pEnqBK6y_ZE2LuwHWR_OAiEA_-RFrFoDVkqYUTf-
0DDnvsU5FT8wqheH4pHbyvqjh_djeDVjgVkB5DCCAeAwggGDoAMCAQICBGwrWPIwDAYIKoZIz
j0EAwIFADBkMQswCQYDVQQGEwJVUzEXMBUGA1UEChMOU3Ryb25nQXV0aCBJbmMxIjAgBgNVBA
sTGUF1dGhlbnRpY2F0b3IgQXR0ZXN0YXRpb24xGDAWBgNVBAMMD0F0dGVzdGF0aW9uX0tleTA
eFw0xOTA3MTgxNzExMjdaFw0yOTA3MTUxNzExMjdaMGQxCzAJBgNVBAYTAlVTMRcwFQYDVQQK
Ew5TdHJvbmdBdXRoIEluYzEiMCAGA1UECxMZQXV0aGVudGljYXRvciBBdHRlc3RhdGlvbjEYM
BYGA1UEAwwPQXR0ZXN0YXRpb25fS2V5MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEMfSGPr
r3xL2fR2iV98mmJjqkiGEMUtzotxHe4sOM1xkeQuEWisjkHiLtYStkAoKKttdCo8DRVN06bKg
DGS2ZhaMhMB8wHQYDVR0OBBYEFDRC0OBwQ401s4_RNGe9fROBHi8YMAwGCCqGSM49BAMCBQAD
SQAwRgIhAO0W0djQrcjEMIshhjgA8vKwx4zRT5WRvCKfZK_YgCorAiEAot3DQBY0y9N_rJOwt
ZYo-yUOpju64X3QzHw10o3oMLloYXV0aERhdGFZATSyyBYoH-
owyRyy_WxSXxKdk4SXIbgPcYuaz8s47lSqOkEAAAAAAAAAAAAAAAAAAAAAAAAAACwMBDVxPO
Z5To939FLGuhTPaaMA1jqTvajZrqWKbnI81yhEndkjQPbL7Q6W5TerIq_rowNstdvrXCLsOw4
aO1-xJB-Q4-WkNPMdYhIiN9yt0rRIiev917ezeNzwIosjrN99MUHR_J_Sw6Js4Q49mllAgZ-
gaxnqd7pmIX_V6B7oDfWaKmvImwxo3pGXqXb-6pboouYVbiMl6WA-
TooklND0pIXWxdp2SvbfkoIur-
c8wClAQIDJiABIVggYuHJDPmHnCV9BJSQoPFp1r05eYfEzavw3JRQzLPg7tYiWCCCLNLY2cui
vtzxnwOSYHhKYOPHMmTcyRW4_Jy2IUZqFA",
        "clientDataJSON":
"eyJ0eXBlIjoid2ViYXV0aG4uY3JlYXRlIiwiY2hhbGxlbmdlIjoiRkNNMHV0SWxwNEt3NG8y
RHB6bnI1USIsIm9yaWdpbiI6Imh0dHBzOi8vc2FrYTIwOS5zdHJvbmdhdXRoLmNvbSJ9"
      },
      "type": "public-key"
    },
    "strongkeyMetadata": {
      "version": "1.0",
      "create_location": "Sunnyvale, CA",
```

```
      "origin": "https://<FQDN>",
      "username": "johndoe"
    }
  }
}
```

Value	Explanation
did	Unique domain identifier for a cryptographic domain.
protocol	The FIDO protocol to be used in this request (FIDO2_0).
authtype	Type of authorization used to request the service. Can be *PASSWORD* or *HMAC*.
svcusername	Username requesting the service. The service credentials are looked up in the 'service' setup of authentication system based on *Lightweight Directory Access Protocol (LDAP)/Active Directory (AD)*.
svcpassword	Password of the service username specified above.
version	The metadata object version. Currently the only supported version is *1.0*.
create_location	The location from which the key is being created. This value will be returned with calls to *getkeyinfo* and can be presented to the user.
username	The user account to register the FIDO key.
origin	The fully qualified origin of the requester.
id	The credential's identifier. The requirements for the identifier are distinct for each type of credential. For example, It might represent a username for username/password tuples.
rawId	The ArrayBuffer contained in the [[identifier]] internal slot.
type	This attribute's getter returns the value of the object's interface object's [[type]] slot, which specifies the credential type represented by this object.
attestationObject	This attribute contains an attestation object, which is opaque to and cryptographically protected against tampering by the client.
clientDataJSON	This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data passed to the Authenticator by the client in order to generate this assertion.

4.2.3.2—FIDO2 v3 API *register* Examples

- If the *FIDO2_0* response is successful, the web service will return status code *200 OK* and a successful response string:

```
{"Response":"Successfully processed registration response"}
```

4.2.4—FIDO2 v3 API *preauthenticate* Mechanics

The *preauthenticatie* operation is used to generate a challenge to be signed by the FIDO Authenticator. The web service receives a set of parameters that contains all the metadata necessary to create a challenge.

Parameter	Explanation
svcinfo	Object that carries FIDO2 v3 API service information.
payload	Object that carries the *preauthenticate* web service parameters. See below for the *FIDO2_0* request body.

4.2.4.1—FIDO2 v3 API *preauthenticate* Request Body

The following request body is sent during the *preauthenticate* call:

- URL: https://<FQDN>:<PORT>/skfs/rest/preauthenticate
- HTTP Method: POST
- *FIDO2_0* request body:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe",
    "options": {}
  }
}
```

Value	Explanation
did	Unique domain identifier for a cryptographic domain.
protocol	The FIDO protocol to be used in this request (FIDO2_0).
authtype	Type of authorization used to request the service. Can be *PASSWORD* or *HMAC*.
svcusername	Username requesting the service. The service credentials are looked up in the 'service' setup of authentication system based on *Lightweight Directory Access Protocol (LDAP)/Active Directory (AD)*.
svcpassword	Password of the service username specified above.
username	The user account to which owns the FIDO key.
options	A JSON containing a list of options to associate with this challenge. The options JSON can be empty, but is otherwise a required parameter.

4.2.4.2—FIDO2 v3 API *preauthenticate* Example

- If the *FIDO2_0* response is successful, the web service will return status code *200 OK* and a the following JSON:

```
{
  "Response": {
        "challenge": "k1YeYZZ6HDmg3ruKinb2SQ",
        "allowCredentials": [{
              "type": "public-key",
              "id": "WLwVfJwmEXQJidjyX_ucXNnJMXyhZeNoVEsZU5MI1t34S-uSTkU-
V77wk52fhui-CNhWIk8po6Awf_pfElCuqQ",
              "alg": -7
        }],
        "rpId": "strongkey.com"
  }
}
```

4.2.5—FIDO2 v3 API *authenticate* Mechanics

The *authentication* operation immediately follows a call to *preauthenticate* and is used to complete authentication of a FIDO Authenticator. The web service receives a set of parameters that contains all the metadata necessary to authenticate the key.

Parameter	Explanation
svcinfo	Object that carries FIDO2 v3 API service information.
payload	Object that carries the *authenticate* web service parameters. See below for the *FIDO2_0* request body.

4.2.5.1—FIDO2 v3 API *authenticate* Request Body

The following request body is sent during the authenticate call:

- URL: https://<FQDN>:<PORT>/skfs/rest/authenticate
- HTTP Method: POST
- *FIDO2_0* request body:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "publicKeyCredential": {
      "id": "WLwVfJwmEXQJidjyX_ucXNnJMXyhZeNoVEsZU5MI1t34S-uSTkU-
V77wk52fhui-CNhWIk8po6Awf_pfElCuqQ",
      "rawId": "WLwVfJwmEXQJidjyX_ucXNnJMXyhZeNoVEsZU5MI1t34S-uSTkU-
V77wk52fhui-CNhWIk8po6Awf_pfElCuqQ",
      "response": {
        "authenticatorData":
```

```
"UoqJLa5UxCjgq_vY4aN3KaBN0YJ8PiXSQAnEkb84jqsBAAABZA",
        "signature": "MEUCIQDcsR-
1lLuIcSIa01RzxtoQMdDBY_PfiEHwfxTprKOVXgIgZcbilg-
3H1agt7xTHLvI45IIz2ReBXmrzi9h1YYr124",
        "userHandle": "",
        "clientDataJSON":
"eyJjaGFsbGVuZ2UiOiJrMVllWVpaNkhEbWczcnVLaW5iMlNRIiwib3JpZ2luIjoiaHR0cHM6
Ly9maWRvMi5zdHJvbmdrZXkuY29tIiwidHlwZSI6IndlYmF1dGhuLmdldCJ9"
      },
      "type": "public-key"
    },
    "strongkeyMetadata": {
      "version": "1.0",
      "last_used_location": "Cupertino, CA",
      "username": "johndoe",
      "origin": "https://<FQDN>"
    }
  }
}
```

Value	Explanation
did	Unique domain identifier for a cryptographic domain.
protocol	The FIDO protocol to be used in this request (FIDO2_0).
authtype	Type of authorization used to request the service. Can be *PASSWORD* or *HMAC*.
svcusername	Username requesting the service. The service credentials are looked up in the 'service' setup of authentication system based on *Lightweight Directory Access Protocol (LDAP)/Active Directory (AD)*.
svcpassword	Password of the service username specified above.
version	The metadata object version. Currently the only supported version is *1.0*.
last_used_location	The location from which the key is being activated. This value will be returned with calls to *getkeyinfo* and can be presented to the user.
username	The user account who owns the FIDO key.
origin	The fully qualified origin of the requester.
id	The credential's identifier. The requirements for the identifier are distinct for each type of credential. For example, It might represent a username for username/password tuples.
rawId	The ArrayBuffer contained in the [[identifier]] internal slot.
type	This attribute's getter returns the value of the object's interface object's [[type]] slot, which specifies the credential type represented by this object.
authenticatorData	This attribute contains the Authenticator data returned by the Authenticator. See §6.1 Authenticator Data.

Value	Explanation
clientDataJSON	This attribute, inherited from AuthenticatorResponse, contains the JSON-serialized client data passed to the Authenticator by the client to generate this assertion.
userHandle	This attribute contains the user handle returned from the Authenticator, or null if the Authenticator did not return a user handle. See §6.3.3 The authenticatorGetAssertion Operation.
signature	This attribute contains the raw signature returned from the Authenticator. See §6.3.3 The authenticatorGetAssertion Operation.

4.2.5.2—FIDO2 v3 API *authenticate* Examples

- On a success, the web service will return the following JSON with a *200 OK"*

```
{
"Response": ""
}
```

4.2.6—FIDO2 v3 API *getkeysinfo POST* Mechanics

The */getkeysinfo POST* operation is used to retrieve a list of all the registered FIDO keys for the user. The web service receives a set of parameters that contains all the metadata necessary to get a specified user's keys.

4.2.6.1—FIDO2 v3 API *getkeysinfo* Request Body

The following request body is sent during the getkeysinfo call:

- URL: https://<FQDN>:<PORT>/skfs/rest/getkeysinfo
- HTTP Method: POST
- *FIDO2_0* request body:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "username": "johndoe"
  }
}
```

4.2.6.2—FIDO2 v3 API *getkeysinfo POST* Request

On a success, the web service will return the following JSON *FIDO2_0* response with a *200 OK*, including a list of registered FIDO keys for the user:


```
{
  "Response": {
        "keys": [{
              "randomid": "1-1-johndoe-702180",
              "randomid_ttl_seconds": "300",
              "fidoProtocol": "FIDO2_0",
              "fidoVersion": "FIDO2_0",
              "createLocation": "Sunnyvale, CA",
              "createDate": 1569459752000,
              "lastusedLocation": "Not used yet",
              "modifyDate": 0,
              "status": "Active",
              "displayName": "johndoe"
        }, {
              "randomid": "1-1-johndoe-701309",
              "randomid_ttl_seconds": "300",
              "fidoProtocol": "FIDO2_0",
              "fidoVersion": "FIDO2_0",
              "createLocation": "Sunnyvale, CA",
              "createDate": 1569459651000,
              "lastusedLocation": "Not used yet",
              "modifyDate": 0,
              "status": "Active",
              "displayName": "johndoe"
        }]
  }
}
```


4.2.7—FIDO v3 API *deregister* Mechanics

The *deregister POST* operation is used to delete a registered FIDO key. The web service receives a set of parameters that contains all the metadata necessary to deregister a user's specified key.

4.2.7.1—FIDO2 v3 API *deregister POST Request*

The following request body is sent during the *deregister* call:

- URL: https://<FQDN>:<PORT>/skfs/rest/deregister
- HTTP Method: POST
- *FIDO2_0* request body:


```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "PASSWORD",
    "svcusername": "svcfidouser",
```

```
    "svcpassword": "Abcd1234!"
  },
  "payload": {
    "keyid": "1-1-johndoe-702180"
  }
}
```

4.2.7.2—FIDO2 v3 API *deregister POST* Request

- On a success, the web service will return the following JSON with a *200 OK"*

```
{
  "Response": "Successfully deleted user registered security key"
}
```

4.2.8—Calling FIDO2 Server

4.2.8.1—Authentication

All the web services available in the v3 API accept HMAC authentication scheme. The credential (secret key/access key) must be stored by the calling application, which uses it to calculate HMAC and send it as part of the authorization header.

Follow the steps below to create the authorization header:

- Create the request body with the required parameters. For example:

```
{
  "svcinfo": {
    "did": 1,
    "protocol": "FIDO2_0",
    "authtype": "HMAC"
  },
  "payload": {
    "username": "johndoe",
    "displayname": "Initial Registration",
    "options": {
      "attestation": "direct"
    },
    "extensions": "{}"
  }
}
```

- Calculate sha256 digest of the body and then base64 encode the value. The base64 body digest is NzIzZTRjYjY2NmQ4YjFmMTkzZjJkN2M5MTA4YzE2NWZmZmVkYWQzMTgwYzgxMjc1NjgyYzNDU0NTJmNDA0ZQ==.

- Calculate HMAC over the concatenation of the following parameters:

```
“HTTP Method” + “base64 body hash” + “contentType/mimetype” + “Current
Date” + “api version” + “HTTP URI”
```

- Example request to HMAC:

```
“POST
ghvgYnepusGvHrXZ5huvDb4Zavi+i2VKK/b3PmHxLoo=
application/json
Wed, 25 Sep 2019 18:12:30 PDT
2.0
/api/domains/1/fidokeys/registration/challenge”
```

4.2.8.2—Calling the Endpoint

Once the Authorization HMAC is calculated, a *challenge* is returned back to the browser.

4.2.8.3—Sample Code

The full sample code to make all the FIDO2 v3 API calls can be found in the PoC application:

- GitHub
  https://github.com/StrongKey/fido2/blob/master/sampleapps/java/poc/server/src/main/java/com/strongkey/poc/SKFSClient.java
- SourceForge
  https://sourceforge.net/projects/strongkeyfido/files/v4.4/sampleapps/poc/

5—Troubleshooting

5.1—Error Codes and Their Meanings

Following is a list of error and message codes the appliance logs for any transaction.

Code	SKCE-ERR-1000
Message	Caught an exception\: {0}
Explanation	Generic message for when an exception is caught by the application. It is followed by the cause of the exception.
Code	SKCE-ERR-1003
Message	NULL or invalid argument\\: {0}
Explanation	Message indicating a null or empty parameter.
Code	SKCE-ERR-1092
Message	SKCE Domain does not exist {0}
Explanation	Indicates the domain ID passed by the application does not exist.
Code	SKCE-ERR-1093
Message	SKCE Domain inactive
Explanation	Indicates that the domain ID passed by the application has been deactivated and cannot be used anymore.
Code	SKCE-ERR-1112
Message	Could not reload SKCE configuration\: {0}
Explanation	When the wizards reload configuration properties on the appliance; this error indicates the configurations could not be loaded successfully.
Code	SKCE-ERR-5001
Message	Signature not verified for db record; {0}
Explanation	Indicates the signature for the database records cannot be verified.
Code	SKCE-ERR-6000
Message	ZMQ Error\: {0}
Explanation	Generic ZeroMQ error followed by an error message.
Code	SKCE-ERR-6001
Message	ZMQ Publisher received an invalid object for publishing\: {0}
Explanation	Indicates the ZMQ publisher receives an invalid object for replication. This can only happen if the database entry for replication has been modified manually to include an invalid object.

Code	SKCE-ERR-6008
Message	ZMQ Subscriber received an unknown object\: {0}
Explanation	The ZMQ subscriber received an invalid object for replication. This can only happen if the database entry for replication has been modified manually to include an invalid object or it is trying to replicate a record for a schema the server does not understand.
Code	SKCE-ERR-6009
Message	ZMQ Subscriber could not parse replicated proto object\: {0}
Explanation	The ZMQ subscriber could not parse the Google Proto buffer object created for the object that is to be replicated.
Code	SKCE-ERR-6010
Message	ZMQ Subscriber received an invalid operation on a nonexistent object\: {0}
Explanation	This error occurs if ZMQ receives an operation which is not allowed on that object. Example: Updating a record that has not been added yet.
Code	SKCE-ERR-6011
Message	ZMQ Subscriber did not add replicated object to database—already exists locally\: {0}
Explanation	This error is generated when the object has already been replicated but the acknowledgment has not reached the original server, and it keeps trying to replicate it over and over again. Indicates that either the ZMQ acknowledger is broken or firewall rules between servers need to be fixed.
Code	SKCE-ERR-6012
Message	ZMQ Subscriber received an invalid operation\: {0}
Explanation	Occurs if ZMQ receives an operation which is not one of ADD | DELETE | UPDATE.
Code	SKCE-ERR-6014
Message	ZMQ BacklogProcessor cannot replicate object—did not find it in local database\: {0}
Explanation	This error is printed if the object that is being replicated has already been deleted and ZMQ backlog processor cannot find it.
Code	SKCE-ERR-6015
Message	ZMQ BacklogProcessor found an invalid object in Replication table for publishing\: {0}
Explanation	Indicates the ZMQ backlog processor found an object in the table that doesn't belong.
Code	SKCE-ERR-6016
Message	ZMQ BacklogProcessor failed to push object\: {0}
Explanation	This is printed if ZMQ backlog processor cannot push an object followed by the error message.
Code	SKCE-ERR-6025
Message	ZMQ Subscriber did not delete object from database—object doesn't exist\: {0}
Explanation	This error is printed if ZMQ subscriber is trying to delete an object that has already been deleted. This can happen if replication is broken and backlog processor has already queued objects multiple times.

Code	SKCE-ERR-6094
Message	ZMQ Error\: Server is NOT Active—replication will NOT be started on {0}
Explanation	This message gets printed if there are server records in the database that are inactive.
Code	SKCE-ERR-6095
Message	ZMQ Error\: Server is NOT an Active Publisher—replication will NOT be started on {0}
Explanation	This message is printed if a specific server is not configured as a publisher.
Code	SKCE-ERR-6096
Message	ZMQ Error\: Server is NOT a Publisher—replication will NOT be started on {0}
Explanation	This message is printed if a specific server is not configured as a publisher.
Code	SKCE-ERR-6099
Message	ZMQ Error\: Server is NOT configured with a SID and/or FQDN—replication will NOT be started on {0}
Explanation	This error is printed if a server FQDN does not match the one configured in the database.
Code	SKCE-ERR-6090
Message	Object already persisted by another thread\: {0}
Explanation	This error is printed if two threads try to persist the same data.
Code	SKCEWS-ERR-3014
Message	NULL argument\: {0}
Explanation	This error indicates a null parameter followed by the name of the parameter.
Code	SKCEWS-ERR-3053
Message	Invalid argument\: {0}
Explanation	This error indicated invalid argument followed with the name of the parameter.
Code	SKCEWS-ERR-3055
Message	Invalid user: {0}
Explanation	This error indicates that the service credential used are invalid.
Code	FIDO-ERR-0001
Message	Caught an exception\: {0}
Explanation	Generic exception message. Followed by the actual error message.
Code	FIDO-ERR-0002
Message	Null or empty input\: {0}
Explanation	Message indicating a *null* or empty parameter.
Code	FIDO-ERR-0003
Message	Error during preregister\: {0}
Explanation	Indicates a generic error during preregister followed by the actual error message.

Code	FIDO-ERR-0004
Message	Input registrationresponse cannot be null or empty\: {0}
Explanation	Occurs when the registration response sent to the server is empty.
Code	FIDO-ERR-0005
Message	Input registration response does not contain needed fields\: {0}
Explanation	This message displays when the registration response does not have all the required fields, e.g., a missing clientdataJSON or missing ID.
Code	FIDO-ERR-0006
Message	User session in-active\: {0}
Explanation	The user waited too long to click on the security key; the session timed out on the server side. In this case, session info is removed from the hash map.
Code	FIDO-ERR-0007
Message	No valid keys registered; please register first (could be db signature verification failure): {0}
Explanation	Authentication was attempted for an account with no registered keys.
Code	FIDO-ERR-0008
Message	Database access error\: {0}
Explanation	The server cannot access the database.
Code	FIDO-ERR-0009
Message	Error during preauth\: {0}
Explanation	A generic error during preauthenticate. Followed by the actual error message.
Code	FIDO-ERR-0010
Message	Input sign response cannot be null or empty\: {0}
Explanation	Indicates when the authentication response sent to the server is empty.
Code	FIDO-ERR-0011
Message	Input signresponse does not contain needed fields\: {0}
Explanation	This is printed when authentication response does not have all the required fields, e.g., a missing clientdataJSON or missing ID.
Code	FIDO-ERR-0014
Message	JSON parsing exception\: {0}
Explanation	Generic exception printed if the input JSON is invalid.
Code	FIDO-ERR-0015
Message	User signature could not be verified\: {0}
Explanation	This error outputs if the database records have been manually modified and the server cannot verify the integrity of these records.

Code	FIDO-ERR-0016
Message	Registration metadata cannot be null\: {0}
Explanation	StrongKey FIDO2 Server requires the applications to provide some extra metadata related to user and location information during the registration operation. This message is output if *reg_metadata* is not included in the input.
Code	FIDO-ERR-0017
Message	Authentication metadata cannot be null\: {0}
Explanation	StrongKey FIDO2 Server requires the applications to some extra metadata related to user and location information during registration operation. This is printed if *auth_metadata* is not included in the input.
Code	FIDO-ERR-0018
Message	Invalid registration metadata\: {0}
Explanation	StrongKey FIDO2 Server requires the applications to some extra metadata related to user and location information during registration operation. This is printed if *reg_metadata* is invalid.
Code	FIDO-ERR-0019
Message	Invalid request metadata\: {0}
Explanation	StrongKey FIDO2 Server requires the applications to some extra metadata related to user and location information during registration operation. This is printed if the *auth_metadata* is invalid
Code	FIDO-ERR-0020
Message	Invalid request parameters\: {0}
Explanation	This message outputs if the request has invalid parameters.
Code	FIDO-ERR-0021
Message	Request parameters cannot be null\: {0}
Explanation	Occurs if the request parameter is *null*.
Code	FIDO-ERR-0022
Message	User key couldn't be fetched based on random ID; or it has been flushed away\: {0}
Explanation	This error is printed if the requested FIDO key information does not exist
Code	FIDO-ERR-0023
Message	Error deleting user key\: {0}
Explanation	Generic error during key deletion. Followed by an error message.
Code	FIDO-ERR-0024
Message	Error updating user LDAP attribute "FIDOKeysEnabled" to \: {0}
Explanation	LDAP cannot be updated during an update operation.

Code	FIDO-ERR-0025
Message	Failed to generate registration challenge \: {0}
Explanation	Generic error creating the preregister response. Followed by an error message.
Code	FIDO-ERR-0026
Message	Failed to update sign counter value\: {0}
Explanation	The server cannot update the Authenticator counter information.
Code	FIDO-ERR-0027
Message	Too long argument \: {0}
Explanation	An input parameter was longer than the allowed value.
Code	FIDO-ERR-0028
Message	Error deactivating user key\: {0}
Explanation	Generic error during key deactivation. Followed by an error message.
Code	FIDO-ERR-0029
Message	Error activating user key\: {0}
Explanation	Generic error during key activation. Followed by an error message.
Code	FIDO-ERR-0030
Message	Invalid Counter received.
Explanation	The Authenticator counter received has not incremented since the last authentication operation.
Code	FIDO-ERR-0031
Message	Invalid User Presence byte received.
Explanation	The user presence byte in the reg/auth response is invalid.
Code	FIDO-ERR-0032
Message	Appid-Origin mismatch.
Explanation	U2F protocol error that displays if the FIDO APPID-FACETID verification fails.
Code	FIDO-ERR-0033
Message	Authorization failed: Invalid service credentials
Explanation	The service credentials provided by the application are invalid.
Code	FIDO-ERR-0034
Message	DB Signature verification failed.
Explanation	The database record fails integrity verification.
Code	FIDO-ERR-0035
Message	The user is not authorized to perform this operation.
Explanation	The service credential is not authorized for FIDO operations.

Code	FIDO-ERR-0037
Message	Invalid username in the FIDO metadata.
Explanation	The username sent in the metadata and the user for which the response is created do not match.
Code	FIDO-ERR-0038
Message	Failed to parse FIDO policy.
Explanation	During start up, the server could not parse the policy configured in the database.
Code	FIDO-ERR-2001
Message	FIDO 2 Error Message \: {0}
Explanation	Generic error message. Followed by the actual error message
Code	FIDO-ERR-2002
Message	Unsupported Argument \: {0}
Explanation	An unsupported argument is passed by the application to the FIDO API.
Code	FIDO-ERR-5001
Message	Null or empty input\: {0}
Explanation	The input parameter sent is either *null* or empty.
Code	FIDO-ERR-5002
Message	Unsupported FIDO protocol version \:
Explanation	The application sent a FIDO protocol which is not U2F or FIDO2_0.
Code	FIDO-ERR-5004
Message	Signature exception occurred \:
Explanation	Generic exception printed followed by the actual error for anything related to creating and verifying signatures
Code	FIDO-ERR-5005
Message	Failed to verify attestation signature \:
Explanation	FIDO2 Server failed to verify the attestation signature during FIDO registration.
Code	FIDO-ERR-5006
Message	Exception \: {0}
Explanation	Generic exception followed by the actual error message.
Code	FIDO-ERR-5008
Message	Un-supported key type \: {0}
Explanation	The attestation certificate used an unsupported key during U2F.
Code	FIDO-ERR-5009
Message	Public key in the attestation certificate is using an illegal curve \: {0}
Explanation	The attestation certificate used an invalid curve during U2F.

Code	FIDO-ERR-5010
Message	Un-supported or in-valid attestation certificate \: {0}
Explanation	The attestation certificate used an invalid curve during U2F.
Code	FIDO-ERR-5011
Message	JSON could not be parsed \: {0}
Explanation	Generic exception printed if the input JSON is invalid.
Code	FIDO-ERR-5012
Message	Fatal error: Challenge from browser data and authentication response do not match \: {0}
Explanation	This is printed if the challenge sent by server during “pre” calls does not match the ones returned in the response.
Code	FIDO-ERR-5013
Message	Error Base64 decoding Browserdata \: {0}
Explanation	The client data JSON returned during FIDO response is an invalid base64.
Code	FIDO-ERR-5014
Message	Invalid Request Type \: {0}
Explanation	The type of request is invalid.
Code	FIDO-ERR-5015
Message	Invalid challenge, Base64 decode failed \: {0}
Explanation	The challenge inside the clientdataJSON returned during the FIDO response is an invalid base64.
Code	FIDOJPA-ERR-1001
Message	NULL argument\: {0}
Explanation	Generic error for a *null* argument. Followed by the parameter name.
Code	FIDOJPA-ERR-1002
Message	Invalid argument\: {0}
Explanation	Generic error for an invalid argument. Followed by the parameter name.
Code	FIDOJPA-ERR-1003
Message	Missing argument\:
Explanation	Generic error for a missing/empty argument. Followed by the parameter name.
Code	FIDOJPA-ERR-1004
Message	Not Implemented Yet\:
Explanation	Generic message to indicate methods that have not yet been implemented.
Code	FIDOJPA-ERR-2001
Message	Username and Key Handle combination exists.
Explanation	The keyhandle returned in the authentication response does not exist for the user.

Code	FIDOJPA-ERR-2002
Message	fkid does not exist.
Explanation	This is printed if the key with primary key (*fkid*) does not exist.
Code	FIDOJPA-ERR-2004
Message	DID and username combination exists.
Explanation	The username does not exist for the domain ID specified by the calling application.
Code	FIDOJPA-ERR-2005
Message	Policy does not exist.
Explanation	This is printed if a FIDO policy does not exist for a domain.
Code	FIDOJPA-ERR-2006
Message	Unable to persist entry to database.
Explanation	Generic message if the server cannot persist a database entry.

5.2—Solutions for Known Issues

Issue #30: *Deploying StrongKey FidoServer ... Remote server does not listen for requests on [localhost:4848]. Is the server up? Unable to get remote commands...*

...when trying to install the Strongkey/fido2 server on AWS Amazon Linux 2 using the guidelines given, at the sixth step.

Default minimal VMs installs with 1GB memory. Deploy a new VM with 4GB of memory and 10GB HDD; GlassFish, MariaDB, and the OS need enough memory to properly install and thus the install may not work with default values.

6—Tutorial

6.1—Requirements

- Node.js 10.x.x+
- SQLite 3.7.17

NOTE: If you are planning to test the client and the server component of this web application on the same computer—designated COMBINED in this document—then make sure you have a browser version that supports FIDO2.

If you plan to use multiple computers to test this web application, the computer on which the server part of this web application is running is designated as APPSERVER while the computer running the browser(s) is designated APPCLIENT.

The computer running the StrongKey FIDO2 Server is designated FIDO2SERVER.

6.1.1—Installing Required Software Components

6.1.1.1—CentOS 8

```
sudo yum install -y gcc-c++ make
sudo yum install nodejs
sudo yum install sqlite
```

6.1.1.2—CentOS 7

Download any version 10.x.x or higher of *node.js* from the following link:
https://nodejs.org/en/download/

Install *node.js* using the instructions from the following link:
https://github.com/nodejs/help/wiki/Installation

```
sudo yum install sqlite
```

6.1.1.3—Ubuntu

```
sudo apt-get update
sudo apt install nodejs
sudo apt install npm
sudo apt install sqlite
```

6.1.1.4—Windows 10

To install *node.js*:

- **Browse** to https://nodejs.org/en/download/
- **Download** the latest Windows Installer
- **Run** the installer and **follow the prompts**

To install SQLite:

- **Browse** to https://www.sqlite.org/download.html.
- Under *Precompiled Binaries for Windows*, **download** the `.zip` that starts with **sqlite-tools-win32-x86...**

  NOTE: Windows users will need to have installed a compression/unpacking application such as 7-zip, WinZip, etc.

- **Extract** the `.zip` file to the desired path
- **Edit** the Windows PATH variable to include the extracted folder that contains the *sqlite3* executable

6.1.1.5—Mac OS

To install node.js:

- **Browse** to https://nodejs.org/en/download/
- **Download** the latest MacOS Installer (`.pkg`)
- **Run** the installer (and **fail**)
- Click **Settings→Security** and **Privacy→General**
- Under *Allow Apps* find the `node...pkg` file and select **Open Anyway**
- Select **Open**

To install SQLite, browse to https://www.sqlite.org/download.html

- **Download** `sqlite-autoconf-[version number].tar.gz`
- **Open a terminal** to the location of the `sqlite-autoconf` file
- **Run the following** commands, replacing *[version number]* with your SQLite version:

```
tar zxvf sqlite-autoconf-[version number].tar.gz
cd sqlite-autoconf-[version number]
./configure --prefix=/usr/local
make
make install
```

6.2—Installing and Deploying the PREFIDO2 Web Application

1. If using a single computer for testing the client and server portions of this tutorial on COMBINED, modify /etc/hosts or C:\Windows\System32\drivers\etc\hosts file (depending on whether you are using Linux/OS-X or Windows) to include fido2tutorial.strongkey.com as an alias for localhost (the entry with 127.0.0.1).

   If you plan to test the tutorial web application with a browser from a different client computer (APPCLIENT) while running the server component of the tutorial web application on APPSERVER, then identify the IP address of your APPSERVER and add the fido2tutorial.strongkey.com alias to APPSERVER's IP address within the hosts file on the APPCLIENT:

   a. CentOS/Ubuntu/Mac

   ```
   sudo vi /etc/hosts
   ```

   b. Windows

   - Run Notepad as administrator.
   - In *Notepad* click File→Open... and edit `c:\Windows\System32\Drivers\etc\hosts`.
   - Add *fido2tutorial.strongkey.com* after *localhost* (uncomment the *localhost* line if necessary).
   - Save the file.
   - Close Notepad.

2. Ping *fido2tutorial.strongkey.com*.

   
   NOTE: If you have a firewall on the APPSERVER/COMBINED, add a rule to open port 3001 so network connections can reach the web application.

   a. CentOS (If you are using Ubuntu, use `apt` instead of `yum`).

   ```
   sudo yum install firewalld
   sudo firewall-cmd --zone=public --add-port=3001/tcp --permanent
   sudo firewall-cmd --complete-reload
   ```

3. **Deploy** the project.
   a. **Download** the *prefido2* source code. If you are using Ubuntu use `apt` instead of `yum`.

      ```
      sudo yum install wget
      wget
      https://github.com/StrongKey/fido2/raw/master/tutorial/node/prefido2.t
      gz
      ```

      If using Windows, download to the following file:

      https://github.com/StrongKey/fido2/raw/master/tutorial/node/prefido2.tgz

   b. **Unzip** the StrongKey FIDO2 Tutorial:

      ```
      tar zxvf prefido2.tgz
      ```

   c. **Change directory** into *prefido2*:

      ```
      cd prefido2/
      ```

   d. **Install** the required node modules.

      ```
      npm install
      ```

   e. **Install** pm2. This is the process manager we will use to run the application:

      ```
      npm install pm2@latest -g
      ```

   f. **For Windows only: install** *node-gyp* and *sqlite3* manually.

      ```
      npm install -g node-gyp
      npm install sqlite3
      ```

   g. **Start** the project.

      ```
      pm2 start main.js
      ```

   h. [OPTIONAL] **Take a snapshot** of your currently running Node applications; this allows *pm2* to restart your application automatically upon restart of *pm2*. Ignore this step on Windows/Mac OS.

      ```
      sudo pm2 startup systemd
      ```

   i. **Take a snapshot** of your currently running Node applications which allows *pm2* to restart your application automatically upon restart of *pm2*.

      ```
      pm2 save
      ```

   j. **Browse** to *https://fido2tutorial.strongkey.com:3001*. When prompted, **add an exception** for a self-signed certificate. Messages will vary by browser.

NOTE: The default certificate is a 1-year certificate.

4. The home page displays. **Register** a new user by clicking **Sign Up**. **Enter** the required registration information and click **Sign Up**.
5. **Login** using the credentials you just registered. The sample application *Quote Boat* displays.
6. Do whatever floats your boat, then **Logout**.

6.3—FIDO2-enabling the PREFIDO2 Web Application

1. **Copy** the PREFIDO2 web application folder (*prefido2*) and **rename** the copy to *postfido2*.

```
cp -r prefido2 postfido2
```

2. **Delete** old users from the database.

```
sqlite3 postfido2/db/aftdb.db
delete from users;
.exit
```

3. **Open** `postfido2/templates/register.html` in your preferred text editor.
   a. **Copy** this snippet of code (between the AAAAA... lines, but NOT including the AAAAA.... lines) and **paste** it between the AAAAA lines in `register.html`, replacing the existing content in the HTML file.

      Here we are removing the action and method attributes of the form to remove the old registration post request. This will be replaced by a call to a function in `functions.js`.

```
<!-- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA -->
<form  id="regform" >
<!-- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA -->
```

   b. **Copy** this snippet of code (between the BBBBB... lines, but NOT including the BBBBB.... lines) and **paste** between the BBBBB lines in `register.html`, replacing the existing content in the HTML file.

      Here we are removing the *passcontainer* and *password* input and replacing it with *id displayname* input, which is used to identify the name of the FIDO2 Token used in registration.

```
<!-- BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB -->
<input class="input-out" type="text" id="displayname"
name="displayname" placeholder="Display Name"><br>
<!-- BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB -->
```

c. **Copy** this snippet of code (between the CCCCC... lines) and **paste** between the CCCCC lines in `register.html`, replacing the existing content in the HTML file.

Here we remove the *type*, *form*, and *value* from the button and add *onclick="submitForm('registration')"*. This will call the function that will be added to `functions.js`.

```
<!-- CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC -->
<button id="regbutton" onclick="submitForm('registration')">Sign
Up</button>
<!-- CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC -->
```

d. **Copy** this snippet of code (between the DDDDD... lines) and **paste** between the DDDDD lines in `register.html`.

```
<!-- DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD -->
<script type="text/javascript" src="/js/common.js"></script>
<!-- DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD -->
```

4. **Open** `postfido2/templates/login.html` in your preferred text editor.

a. **Copy** this snippet of code (between the EEEEE... lines) and **paste** between the EEEEE lines in `login.html`, replacing the existing content in the HTML file.

Here we remove the action and method aspects of the form to remove the old login post request. This will be replaced by a call to a function in `functions.js`.

```
<!-- EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE -->
<form id="loginform">
<!-- EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE -->
```

b. **Delete** the snippet of code (between the FFFFF... lines). Here we delete the *passcontainer* and *password* input. They are no longer needed, thanks to FIDO2!

```
<!-- FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF -->

<!-- FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF -->
```

c. **Copy** this snippet of code (between the GGGGG... lines) and **paste** between the GGGGG lines in `login.html`, replacing the existing content in the HTML file.

Here we remove the *type*, *form*, and *value* attributes from the button and add *onclick="submitForm('authentication')"*. This will call the function that will be added to to `functions.js`.

```
<!-- GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG -->
<button onclick="submitForm('authentication')">Sign In</button>
<!-- GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG -->
```

d. **Copy** this snippet of code (between the HHHHH... lines) and **paste** between the HHHHH lines in `register.html`.

```
<!-- HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH -->
<script type="text/javascript" src="/js/common.js"></script>
<!-- HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH -->
```

5. Open `postfido2/templates/js/functions.js` in your preferred text editor.
   a. Copy this snippet of code (between the IIII... lines) and paste between the IIII lines in `functions.js`.

      Here we add functions to the APPCLIENT that are used to request challenges from the APPSERVER, pass the challenges to the FIDO2 Token, and submit challenge results to the APPSERVER.

```
//IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
function submitForm(intent){
  if(intent=="registration"){
  $.post('/getChallenge', {
      'intent' : intent,
      'username': $('#regusername').val(),
      'displayname': $('#displayname').val(),
      'firstname': $('#firstname').val(),
      'lastname': $('#lastname').val()
  }).done(resp => {
     if(resp.Response == "sqlite-error"){
       console.log(resp.Response);
       location.reload();
     }else if(resp.Response == "skfs-error"){
       console.log(resp.Response);
     } else {
       document.getElementById("failed").style.display = "none";
       document.getElementById("failedbreak").style.display = "none";
       callFIDO2Token(intent,resp.Response);
     }

      }).fail((jqXHR, textStatus, errorThrown) => {
     alert(jqXHR, textStatus, errorThrown);
   });

} else if(intent=="authentication")
   $.post('/getChallenge', {
     'intent' : intent,
       'username': $('#username').val()
   })
       .done((resp) => {
         if(!
resp.Response.toString().toLowerCase().includes("error")){
           callFIDO2Token(intent,resp.Response);
         } else {
           alert("Username not registered");
         }

       })
       .fail((jqXHR, textStatus, errorThrown) => {
         alert(jqXHR, textStatus, errorThrown);
       });

}
```

```
 function callFIDO2Token(intent,challenge) {
   let challengeBuffer = challengeToBuffer(challenge);
   let credentialsContainer = window.navigator;
   if(intent=="registration"){
       credentialsContainer.credentials.create({ publicKey:
challengeBuffer.Response })
      .then(credResp => {
      let credResponse = responseToBase64(credResp);
      credResponse.intent = intent;
           $.post('/submitChallengeResponse',  credResponse)
               .done(regResponse => onResult(intent,regResponse))
               .fail((jqXHR, textStatus, errorThrown) => {
                   console.log(jqXHR, textStatus, errorThrown);
               });
          })
      .catch(error => {
          alert(error);
      });
     } else if (intent=="authentication"){
       credentialsContainer.credentials.get({ publicKey:
challengeBuffer.Response })
      .then(credResp => {
          let credResponse = responseToBase64(credResp);
          credResponse.intent = intent;
          $.post('/submitChallengeResponse', credResponse)
            .done(authResponse => onResult(intent,authResponse))
            .fail((jqXHR, textStatus, errorThrown) => {
                alert(jqXHR, textStatus, errorThrown);
            });
      })
      .catch(error => {
          alert(error);
      });
   }
 }
   function onResult(intent,response){
     if(intent=="registration"){
       if(!
response.Response.toString().toLowerCase().includes("error")){
    window.location.replace(window.location.protocol + "//" +
window.location.host + "/login");
       } else {
    alert(response.Response);
       }
   } else if(intent=="authentication"){

     if(response.Response.toString() == "{\"Response\":\"\"}"){
       window.location.replace(window.location.protocol + "//" +
window.location.host + "/dashboard");
     } else {
       alert(response.Response);
     }
   }
```

```
  }
//IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
```

6. **Create** the file `postfido2/templates/js/common.js`.
   a. **Copy** this snippet of code (between the JJJJ... lines) and **paste** between the JJJJ lines in `postfido2/templates/js/common.js`. This code is used for encoding and decoding data to and from the FIDO2 Token.

```
//JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
function challengeToBuffer(input) {
    input = JSON.parse(input);
    input.Response.challenge = decode(input.Response.challenge);
    if(typeof input.Response.user !== 'undefined'){
     input.Response.user.id = decode(input.Response.user.id);
    }


if (input.Response.excludeCredentials) {
for (let i = 0; i < input.Response.excludeCredentials.length; i++) {
    input.Response.excludeCredentials[i].id =
input.Response.excludeCredentials[i].id.replace(/\+/g,
'-').replace(/\//g, '_').replace(/=/g, '');
    input.Response.excludeCredentials[i].id =
decode(input.Response.excludeCredentials[i].id);
}
}
if (input.Response.allowCredentials) {
for (let i = 0; i < input.Response.allowCredentials.length; i++) {
    input.Response.allowCredentials[i].id =
input.Response.allowCredentials[i].id.replace(/\+/g,
'-').replace(/\//g, '_').replace(/=/g, '');
    input.Response.allowCredentials[i].id =
decode(input.Response.allowCredentials[i].id);
}
}
return input;

}

function responseToBase64(input) {

let copyOfDataResponse = {};
copyOfDataResponse.id = input.id;
copyOfDataResponse.rawId = encode(input.rawId);
if(typeof input.response.attestationObject !== 'undefined'){
  copyOfDataResponse.attestationObject =
encode(input.response.attestationObject);
}
if(typeof input.response.authenticatorData !== 'undefined'){
  copyOfDataResponse.authenticatorData =
encode(input.response.authenticatorData);
  copyOfDataResponse.signature = encode(input.response.signature);
  copyOfDataResponse.userHandle = encode(input.response.userHandle);
```

```
 }
 copyOfDataResponse.clientDataJSON =
encode(input.response.clientDataJSON);
 copyOfDataResponse.type = input.type;
 return copyOfDataResponse;
 }

 let chars =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_';
    let lookup = new Uint8Array(256);
    for (let i = 0; i < chars.length; i++) {
          lookup[chars.charCodeAt(i)] = i;
 }
 let encode = function (arraybuffer) {
    let bytes = new Uint8Array(arraybuffer),
          i, len = bytes.length, base64url = '';
    for (i = 0; i < len; i += 3) {
          base64url += chars[bytes[i] >> 2];
          base64url += chars[((bytes[i] & 3) << 4) | (bytes[i + 1] >>
4)];
          base64url += chars[((bytes[i + 1] & 15) << 2) | (bytes[i +
2] >> 6)];
          base64url += chars[bytes[i + 2] & 63];
    }
    if ((len % 3) === 2) {
          base64url = base64url.substring(0, base64url.length - 1);
    } else if (len % 3 === 1) {
          base64url = base64url.substring(0, base64url.length - 2);
    }
    return base64url;
 };
 let decode = function (base64string) {
    let bufferLength = base64string.length * 0.75,
          len = base64string.length, i, p = 0,
          encoded1, encoded2, encoded3, encoded4;
    let bytes = new Uint8Array(bufferLength);
    for (i = 0; i < len; i += 4) {
          encoded1 = lookup[base64string.charCodeAt(i)];
          encoded2 = lookup[base64string.charCodeAt(i + 1)];
          encoded3 = lookup[base64string.charCodeAt(i + 2)];
          encoded4 = lookup[base64string.charCodeAt(i + 3)];
          bytes[p++] = (encoded1 << 2) | (encoded2 >> 4);
          bytes[p++] = ((encoded2 & 15) << 4) | (encoded3 >> 2);
          bytes[p++] = ((encoded3 & 3) << 6) | (encoded4 & 63);
    }
    return bytes.buffer
 };
 //JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
```

7. **Create** the file `postfido2/constants.js`.

   a. **Copy** this snippet of code (between the KKKK... lines) and **paste** between the KKKK lines in `postfido2/constants.js`. The current values of SKFS_HOSTNAME, SVCUSERNAME, and SVCPASSWORD default to pointing to a FIDO2SERVER hosted by StrongKey. If you are using your own SKFS, replace the example values for SKFS with the *hostname* of the FIDO2SERVER and the *svcusername* and *svcpassword* with the username and password used to access the FIDO2SERVER.

```
//KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
const DID = 1;
const PROTOCOL = "FIDO2_0";
const AUTHTYPE = "PASSWORD";
const SVCUSERNAME = "svcfidouser";
const SVCPASSWORD = "Abcd1234!";

exports.SKFS_HOSTNAME = "demo4.strongkey.com";
exports.SKFS_PORT="8181";
exports.SVCINFO = {
    did: DID,
    protocol: PROTOCOL,
    authtype: AUTHTYPE,
    svcusername: SVCUSERNAME,
    svcpassword: SVCPASSWORD
       };



exports.SKFS_PREAUTHENTICATE_PATH = '/skfs/rest/preauthenticate'
exports.SKFS_AUTHENTICATE_PATH = '/skfs/rest/authenticate'
exports.SKFS_PREREGISTRATION_PATH = '/skfs/rest/preregister'
exports.SKFS_REGISTRATION_PATH = '/skfs/rest/register'
exports.SKFS_GET_KEYS_INFO_PATH = '/skfs/rest/getkeysinfo'
exports.SKFS_DEREGISTER_PATH = '/skfs/rest/deregister'

exports.METADATA_VERSION = "1.0"
exports.METADATA_LOCATION = "Cupertino, CA"
//KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
```

8. **Open** `postfido2/routes.js` in your preferred text editor.

   a. **Copy** this snippet of code (between the LLLL... lines) and **paste** between the LLLL lines in `routes.js`. Here we add include the HTTPS module and the constants file that will be used to call the FIDO2SERVER.

```
//LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
L
 const https = require('https');
 const CONSTANTS = require('./constants');


//LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL
```

b. **Copy** this snippet of code (between the MMMM... lines) and **paste** between the MMMM lines in `routes.js`, replacing the existing content in the JavaScript file. Replace the */loginSubmit* post listener */registerSubmit* with */getChallenge* and */submitChallengeResponse* listeners. The */getChallenge* listener is used to request challenges from the FIDO2SERVER to send to the APPCLIENT for registration and authentication. The */submitChallengeResponse* listener is used to send the responses received from the FIDO2 Token, sent by the APPCLIENT, to the FIDO2SERVER.

```
//MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
 router.post("/getChallenge", (req,res) =>{
   var intent = req.body.intent;
     var username = req.session.username = req.body.username;
 if(intent=="authentication"){
   if(username == ""){
     res.redirect("/login");
     return;
   }
   var db = getDB();
   db.get(`select * from users where username = ? `,[username],
      (err, row) => {
      if (err) {
        log("ERROR: "+ err.message);
      }
      if (row) {
    req.session.possibleuserid = row.id;
     process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
     const data = JSON.stringify({
       svcinfo: CONSTANTS.SVCINFO,
       payload: {
      username: username,
      options: "{}"
       }
     });
     const options = {
       hostname: CONSTANTS.SKFS_HOSTNAME,
       port: CONSTANTS.SKFS_PORT,
       path: CONSTANTS.SKFS_PREAUTHENTICATE_PATH,
       method: 'POST',
       headers: {
    'Content-Type': 'application/json',
    'Content-Length': data.length
       }
     };
     const fido2Req = https.request(options, fido2Res => {
       log(`statusCode: ${fido2Res.statusCode}`);

       fido2Res.on('data', d => {
    log("challengeBuffer=");
    log(d);
    res.json({Response:d.toString()});
       })
     });
```

```
    fido2Req.on('error', error => {
      log(error);
      res.json({Response:"skfs-error"});
    });
    fido2Req.write(data);
    fido2Req.end();
      } else {
   res.json({Response:"sqlite-error"});
      }
    });
} else if(intent=="registration"){
  var firstname = req.session.firstname = req.body.firstname;
  var lastname = req.session.lastname = req.body.lastname;
  var displayname = req.session.displayname= req.body.displayname;
  if(username == "" | displayname=="" | firstname == "" | lastname ==
""){
    res.redirect("/register");
    return;
  }
  var db = getDB();
  db.get(`select * from users where username = ? `,[username],
     (err, row) => {
     if (err) {
       log("ERROR: "+ err.message);
     }
     if (!row) {

 process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
const data = JSON.stringify({
  svcinfo: CONSTANTS.SVCINFO,
   payload: {
      username: username,
      displayname: displayname,
      options: "{\"attestation\":\"direct\"}",
      extensions: "{}"
    }
});
const options = {
  hostname: CONSTANTS.SKFS_HOSTNAME,
  port: CONSTANTS.SKFS_PORT,
  path: CONSTANTS.SKFS_PREREGISTRATION_PATH,
  method: 'POST',
  headers: {
'Content-Type': 'application/json',
'Content-Length': data.length
  }
};
const fido2Req = https.request(options, fido2Res => {
  log(`statusCode: ${fido2Res.statusCode}`);

  fido2Res.on('data', d => {
log("challengeBuffer=");
log(d);
```

```
res.json({Response:d.toString()});
  })
});
fido2Req.on('error', error => {
  log(error);
  res.json({Response:"skfs-error"});
});
fido2Req.write(data);
fido2Req.end();
  } else {
failedRegistration=true;
res.json({Response:"sqlite-error"});
  }
});
}
});

  router.post("/submitChallengeResponse", (req,res) =>{
    var intent = req.body.intent;
      var username = req.session.username;
      var credResponse = req.body;
      var reqOrigin = req.get('host');

  let data = {};
  let path = "";
if(intent=="authentication"){
  var metadataString = JSON.stringify({
  version: CONSTANTS.METADATA_VERSION,
  last_used_location: CONSTANTS.METADATA_LOCATION,
  username: username,
  origin: "https://"+reqOrigin
  });
  var responseString =   JSON.stringify({
  id: credResponse.id,
  rawId: credResponse.rawId,
  response: {
      authenticatorData: credResponse.authenticatorData,
      signature: credResponse.signature,
      userHandle: credResponse.userHandle,
      clientDataJSON: credResponse.clientDataJSON
  },
  type: "public-key"});
data = JSON.stringify({
  svcinfo: CONSTANTS.SVCINFO,
  payload: {
      metadata: metadataString,
      response: responseString
  }
 });
 path = CONSTANTS.SKFS_AUTHENTICATE_PATH;
} else if(intent=="registration"){
  var firstname = req.session.firstname;
  var lastname = req.session.lastname;
```

```
  var db = getDB();
  var metadataString = JSON.stringify({
  version: CONSTANTS.METADATA_VERSION,
  create_location: CONSTANTS.METADATA_LOCATION,
  username: username,
  origin: "https://"+reqOrigin
  });
 var responseString =   JSON.stringify({
    id: credResponse.id,
    rawId: credResponse.rawId,
    response: {
   attestationObject: credResponse.attestationObject,
   clientDataJSON: credResponse.clientDataJSON
    },
    type: "public-key"});

 data = JSON.stringify({
  svcinfo: CONSTANTS.SVCINFO,
    payload: {
     metadata: metadataString,
     response: responseString,
   }
 });
 path = CONSTANTS.SKFS_REGISTRATION_PATH;
}
 const options = {
  hostname: CONSTANTS.SKFS_HOSTNAME,
  port: CONSTANTS.SKFS_PORT,
  path: path,
  method: 'POST',
  headers: {
 'Content-Type': 'application/json',
 'Content-Length': data.length
  }
 };

 const fido2Req = https.request(options, fido2Res => {
  log(`statusCode: ${fido2Res.statusCode}`);

  fido2Res.on('data', d => {
 if(d.toString().toLowerCase().includes("error")){
  res.json({Response:d.toString()});
  return;
 }
    if(intent == "registration"){
      db.run('insert into users(username,first_name,last_name)
values(?,?,?)',[username,firstname,lastname], function(err) {
       if (err) {log("ERROR: "+ err.message);}
           log("user added: \nfirst name: "+firstname+"\nlast name:
"+lastname+"\nusername: "+username);
           req.session.justReg=true;
           log(d);
           res.json({Response:d.toString()});
```

```
          });
      } else if(intent == "authentication"){
    req.session.userid = req.session.possibleuserid;
    log(d);
    res.json({Response:d.toString()});
      }
  })

});

fido2Req.on('error', error => {
  log(error);
  res.json({Response:"error"});
});
process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
fido2Req.write(data);
fido2Req.end();
  });

//MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

c. Copy this snippet of code (between the NNNN... lines) and paste between the NNNN lines in `routes.js`, replacing the existing content in the JavaScript file.

Modify the *deleteUser* listener by replacing the code below with two post requests to FIDO2SERVER. The first post request is to */skfs/rest/getkeysinfo*, which is used to retrieve the user's FIDO2 Token's *keyid*. The second request is to */skfs/rest/deregister*, which deletes the FIDO2 Token registration information. This deletes the user's FIDO2 Token from the FIDO2SERVER's database at the same time the user's info is deleted from the APPSERVER database.

```
//NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
  var username = req.session.username;
  req.session.userid=null;
  log("logout user " +id +" "+username);

   db.run(`delete from users where id = ?`, id, function(err) {
       if (err) {
    return console.error(err.message);
       }
       process.env["NODE_TLS_REJECT_UNAUTHORIZED"] = 0;
  const data = JSON.stringify({
    svcinfo: CONSTANTS.SVCINFO,
    payload: {
   username: username
    }
  });
  const options = {
    hostname: CONSTANTS.SKFS_HOSTNAME,
    port: CONSTANTS.SKFS_PORT,
    path: CONSTANTS.SKFS_GET_KEYS_INFO_PATH,
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
```

```
      'Content-Length': data.length
    }
  };
  const fido2Req = https.request(options, fido2Res => {
    log(`statusCode: ${fido2Res.statusCode}`);

    fido2Res.on('data', d => {
      log("keyInfo=");
      log(d);
      const dataDel = JSON.stringify({
   svcinfo: CONSTANTS.SVCINFO,
   payload: {
       "keyid": JSON.parse(d).Response.keys[0].randomid
   }
      });
      const optionsDel = {
   hostname: CONSTANTS.SKFS_HOSTNAME,
   port: CONSTANTS.SKFS_PORT,
   path: CONSTANTS.SKFS_DEREGISTER_PATH,
   method: 'POST',
   headers: {
     'Content-Type': 'application/json',
     'Content-Length': dataDel.length
   }
      };
      const fido2ReqDel = https.request(optionsDel, fido2ResDel => {
   log(`statusCode: ${fido2ResDel.statusCode}`);

   fido2ResDel.on('data', dDel => {
     log(dDel);
     log("deleted user " +id);
     req.session.justUserDeleted = true;
     res.redirect("/login");
   })
      });
      fido2ReqDel.on('error', errorDel => {
   log(errorDel);
   res.json({Response:"skfs-error"});
      });
      fido2ReqDel.write(dataDel);
      fido2ReqDel.end();
    })
  });
  fido2Req.on('error', error => {
    log(error);
    res.json({Response:"skfs-error"});
  });
  fido2Req.write(data);
  fido2Req.end();
   });
//NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

9. **Deploy** the POSTFIDO2 web application. Open a terminal and navigate to the `/postfido2` directory. **Run** the following commands:
   a. **CentOS/Ubuntu.**
   ```
   pm2 delete main
   pm2 start main.js
   [OPTIONAL] sudo pm2 startup systemd
   pm2 save
   tail -f log
   ```
   b. **Windows/Mac.**
   ```
   pm2 delete main
   pm2 start main.js
   pm2 save
   ```
   c. **Run** this command if the website fails to start:
   ```
   node main.js
   ```
10. **Browse** to *https://fido2tutorial.strongkey.com:3001*.

7—Using SKFS

7.1—Testing the SKFS Cluster with a Sample Web App

To test the cluster with a sample web application, provision the fourth VM to install the sample application and follow the steps here to install the StrongKey *Proof of Concept (PoC)* Java Application. When installing the PoC application, make sure that you follow the steps to NOT install it with a FIDO2 server on the VM; because you already have a FIDO2 server cluster setup following this document, there is no need for an additional FIDO2 server.

The StrongKey PoC Java application is a self-contained web application that demonstrates the use of SKFS for registering users with FIDO2 and U2F Authenticators and, once registered, authenticating them with those Authenticators. The PoC web application also showcases a key management panel where registered users may add Authenticator keys to their account.

1. Login to the PoC VM as *strongkey* upon completing its installation.
2. Using a text editor such as *vi*, modify the application's configuration properties to point the application to the HAProxy load balancer setup in the earlier section of this document:

   ```
   shell> vi /usr/local/strongkey/poc/etc/poc.properties
   ```

3. Change the value of the `property poc.cfg.property.apiuri` and point it to the new load balancer, replacing the `<load-balancer-FQDN>` parameter with the FQDN of the HAProxy VM from your environment:

   ```
   poc.cfg.property.apiuri=https://<load-balancer-FQDN>/api
   ```

4. Run the `certimport.sh` script to import the load balancer's self-signed certificate into the Payara application server running on the PoC VM:

   ```
   shell> /usr/local/strongkey/bin/certimport.sh <load-balancer-FQDN> -p443
   -kGLASSFISH
   ```

5. Restart the Payara application server:

   ```
   shell> sudo service glassfishd restart
   ```

6. Open a browser to the appropriate URL to access the PoC application on the PoC VM, replacing the `<PoC-VM-FQDN>` with the FQDN of the VM on which the PoC application is installed:

   ```
   https://<PoC-VM-FQDN>:8181
   ```

7.2—Prerequisites

It's worth re-stating that the following are required to successfully use the FIDO2 server:

- A FIDO® Certified U2F Authenticator; check this list for vendors of compatible FIDO2 devices
- A browser release that supports the WebAuthn protocol
- A computer with Microsoft Windows, Apple OS-X, or CentOS Linux installed

7.2.1—Linux Users

Linux PCs require the following task to be completed before beginning the demo:

1. Using *sudo*, **modify** the `/etc/udev/rules.d/70-u2f.rules` file. If it doesn't already exist, create it.
2. **Add the following text** to the file:

```
ACTION!="add|change", GOTO="u2f_end", KERNEL=="hidraw*",
SUBSYSTEM=="hidraw", ATTRS{idVendor}=="*", ATTRS{idProduct}=="*",
TAG+="uaccess", LABEL="u2f_end"
```

3. **Reboot** the Linux PC.

7.3—Proof of Concept Java Application

This project is a service provider web application written in JavaScript and Java to work with StrongKey's FIDO® Certified FIDO2 Server, Community Edition.

Web application developers worldwide face multiple challenges in the near future: learning about FIDO2, coding in FIDO2, demonstrating to decision makers what FIDO2 can do for their company, and acquiring budgets and resources to transition to FIDO2 strong authentication. Unless you spend many weeks (or months) understanding how FIDO2 works, addressing all these challenges remains daunting.

StrongKey has released this project to the open-source community to address these challenges. The FIDO2 server allows developers to do the following:

- Setup a FIDO2-enabled single-page web application that can run unmodified and demonstrate FIDO2 registration, authentication, and some simple FIDO2 key management on the client side
- Substitute the stock graphics and logo with your own company's graphics and logo without additional programming—just replace the graphic image files and reload the application; this allows you to demonstrate to peers and management what FIDO2 can do for the company, and how the user experience (UX) might look in its basic form
- Learn how FIDO2 works; all the code is available here in a web application framework
- Use the FIDO® Certified, open-source FIDO2 server with your web application without having to anticipate deployment issues—you will have already deployed this FIDO2 server proof of concept

While this web application can show you how to use W3C's WebAuthn (a subset of the FIDO2 specification) JavaScript, it is also intended to demonstrate how to use FIDO2 protocols with SKFS to enable strong authentication. Follow the instructions below to install this sample.

7.3.1—Prerequisites

This service provider web application example must have a means of connecting with a StrongKey FIDO Server. You can install SKFS either on the same machine as your service provider web application or a different one.

You must have a Java web application server. These instructions assume you are using Payara (GlassFish).

The instructions assume the default ports for all the applications installed; Payara runs HTTPS on port 8181 by default, so make sure all firewall rules allow that port to be accessible.

7.3.2—Installation Instructions on a Server with a FIDO2 Server on a SEPARATE Server

If installing this sample application on a separate server, StrongKey's software stack must be installed to make it work. Follow these steps to do so:

- **Complete** *Steps 1–5* of the FIDO server installation instructions but come back here after completing *Step 5*.
- **Edit** the `install-skfs.sh` script in a text editor; on the line where you see **INSTALL_FIDO=Y** change the value of **Y** to **N**
- **Run** the script `install-skfs.sh`

```
sudo ./install-skfs.sh
```

Continue the installation as shown below in the Installation Instructions on a server with a FIDO2 Server on the SAME server section. Note that this assumes that the FIDO2 Server was previously installed on the server without modifying the `install-skfs.sh` script.

7.3.3—Installation Instructions on a Server with a FIDO2 Server on the SAME Server

1. Create the following directories to configure the WebAuthn servlet home folder.
   ```
   sudo mkdir -p /usr/local/strongkey/poc/etc
   ```
2. Create a configuration file for the service provider web application.
   ```
   sudo vi /usr/local/strongkey/poc/etc/poc-configuration.properties
   ```
3. Fill in the appropriate values (listed in **[ ]**) to configure the sample application with SKFS and an email server (you can also use GMail as the mail server with your own GMail account to send emails. Just make sure you enable access through the Google account's security settings). If the mail server has a self-signed certificate, make sure to import it in the GlassFish TrustStore before continuing.
   ```
   poc.cfg.property.apiuri=https://**[hostname of FIDO Server]**:8181
   poc.cfg.property.mailhost.type=**[SendMail or SSL or StartTLS]**
   poc.cfg.property.mailhost=**[localhost or hostname of mailhost]**
   poc.cfg.property.mail.smtp.port=**[25 (SendMail) or mail server's port]**
   poc.cfg.property.smtp.from=**[local-part of email address]**
   poc.cfg.property.smtp.fromName=**[Human readable name associated with
   email]**
   poc.cfg.property.smtp.auth.user=**[Username used to login to mail
   server]**
   poc.cfg.property.smtp.auth.password=**[Password used to login to mail
   server]**
   poc.cfg.property.email.subject=Verify your email address
   poc.cfg.property.email.type=HTML
   ```
   Save and exit.
4. Download the service provider web application distribution `pocserver-v1.0-dist.tgz`.
   ```
   wget https://github.com/StrongKey/fido2/raw/master/sampleapps/java/poc/
   server/pocserver-v1.0-dist.tgz
   ```
5. Extract the downloaded file to the current directory:
   ```
   tar xvzf pocserver-v1.0-dist.tgz
   ```
6. Execute the `install-pocserver.sh` script as follows:
   ```
   sudo ./install-pocserver.sh
   ```
7. Test that the servlet is running by executing the following cURL command and confirming that you get the API *Web Application Definition Language (WADL)* file back in response:
   ```
   curl -k https://localhost:8181/poc/fido2/application.wadl
   ```

At this point, the PoC server is installed. Continue to install the front-end Angular application.

8. Switch to (or login as) the *strongkey* user. The default password for the *strongkey* user is ShaZam123.

   ```
   su - strongkey
   ```

9. Download the web application distribution for the FIDO2 Server `poc-ui-dist.tar.gz`.

   ```
   wget
   https://github.com/StrongKey/fido2/raw/master/sampleapps/java/poc/angular/
   poc-ui-dist.tar.gz
   ```

10. Extract the downloaded file.

    ```
    tar xvzf poc-ui-dist.tar.gz
    ```

11. Copy all the files to the Payara *docroot*.

    ```
    cp -r dist/*
    /usr/local/strongkey/payara41/glassfish/domains/domain1/docroot
    ```

12. Optional: Modify the background image and the logo image.

    ```
    cp <your background>
    /usr/local/strongkey/payara41/glassfish/domains/domain1/docroot/assets/app
    /media/img/bg/background.jpg
    cp <your logo>
    /usr/local/strongkey/payara41/glassfish/domains/domain1/docroot/assets/app
    /media/img/logo/logo.png
    ```

13. The application is deployed in *docroot* on the PoC server and can be accessed as follows in a browser:

    ```
    https://<FQDN-of-PoC-server>:8181/
    ```

7.3.4—Removal

To uninstall the service provider sample web application, follow the uninstall instructions in the FIDO2 Server, Community Edition Installation Guide. Removing SKFS also removes the sample service provider web application and sample WebAuthn client. If this PoC was installed on top of SKFS, the cleanup script will erase SKFS as well. If this was a standalone install, the cleanup script will only remove the PoC application.

7.3.5—Contributing to the Sample Service Provider Web Application

If you would like to contribute to the sample service provider web application project, please read `CONTRIBUTING.md`, then sign and submit the *Contributor License Agreement (CLA)*.

7.3.6—More Information on FIDO2

For detailed information on the FIDO2 project, visit the technical specification:

- Complete WebAuthn specification
- A useful diagram of WebAuthn functional flow

For more information on the originating jargon and related terms, visit the *Internet Engineering Task Force (IETF) Request for Comments (RFC)*: The definition of "service provider" is in the second paragraph of 1.1. Background, therein referred to as, "relying parties."

7.3.7—Licensing

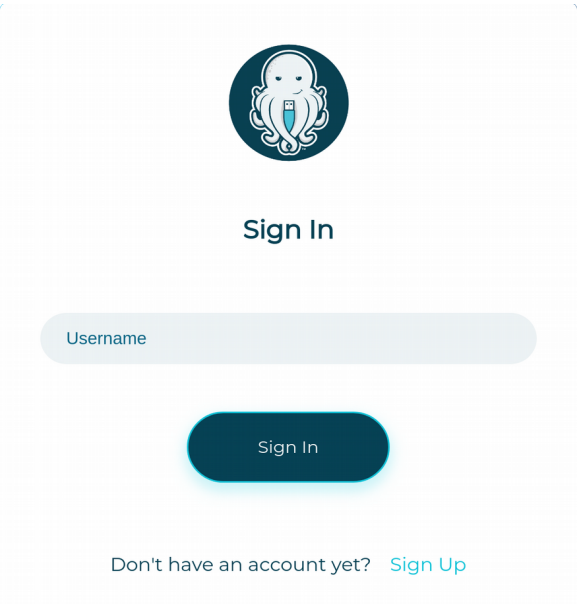
This project is currently licensed under the GNU Lesser General Public License v2.1.

7.4—FIDO2 Registration

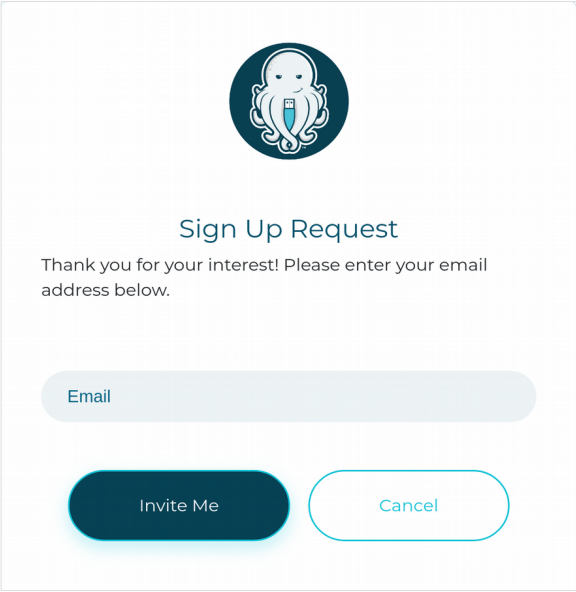
This section explains how to create a new account on the FIDO2 Server.

NOTE: This document refers to this process as “registering a FIDO device.” This is to indicate the generation of a new and unique cryptographic key pair and the public key of that pair being registered with the website.

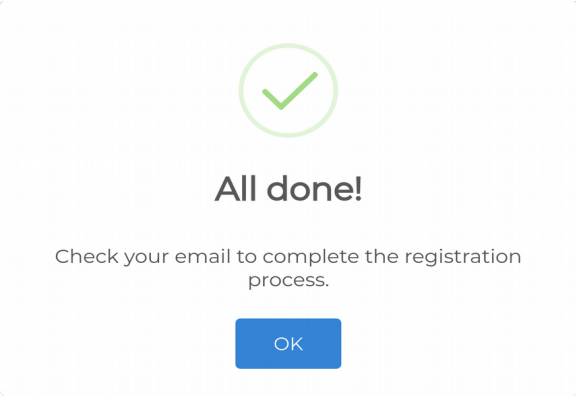
1. In a browser **connect to the FIDO2 Server**.
2. From the home page, click **Sign Up**.



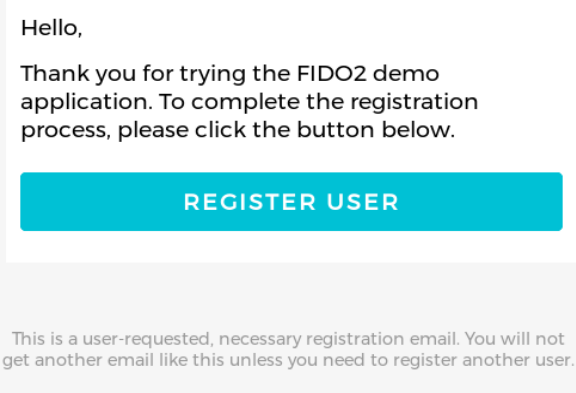
3. A self-invitation panel opens. Supply an email address, then click **Invite Me**.



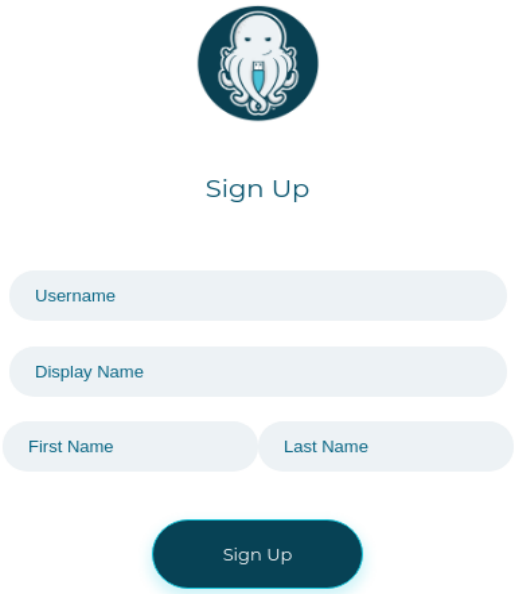
4. A confirmation dialog appears. Click **OK** and open your email client.



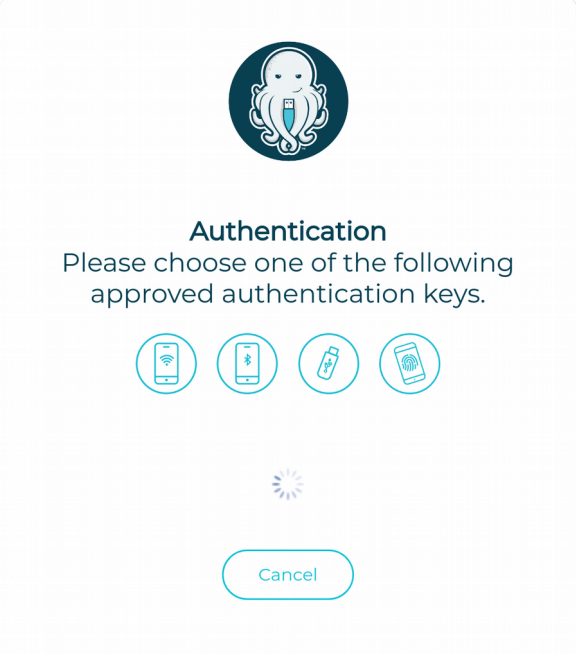
5. Check the email, then click **REGISTER USER** to create the account:



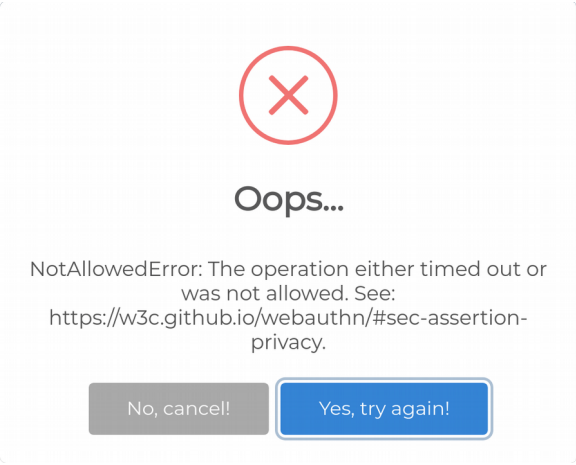
6. Fill out the desired credentials and click **Sign Up**.

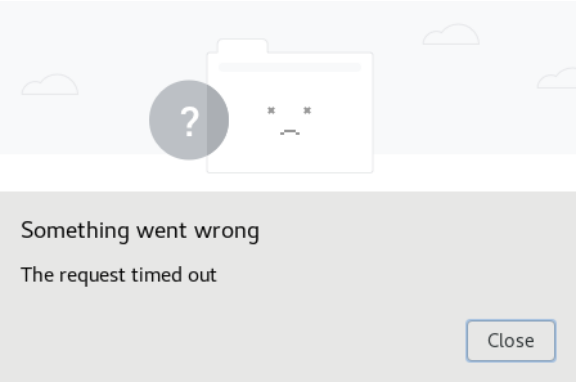


7. You are prompted to **insert your FIDO device and touch it** to verify user presence.

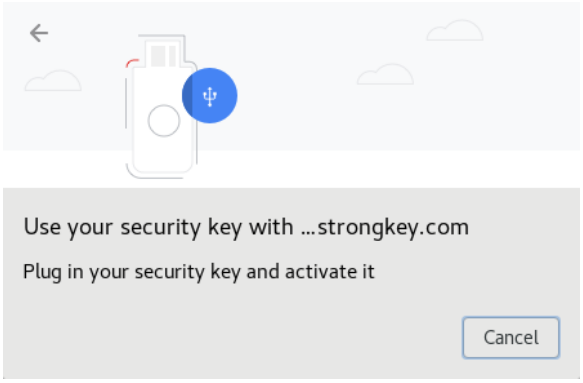


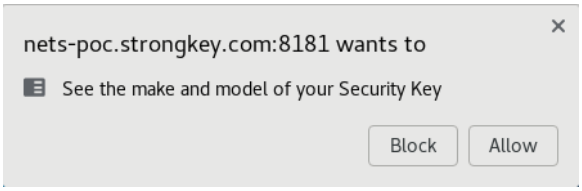
If user presence is not verified within a few minutes, a dialog asks if you would like to try again. Clicking **Yes, try again!** prompts again for a FIDO device. Clicking *No, cancel!* returns to the login panel. A browser dialog may also display; if so, click **Close** to continue.



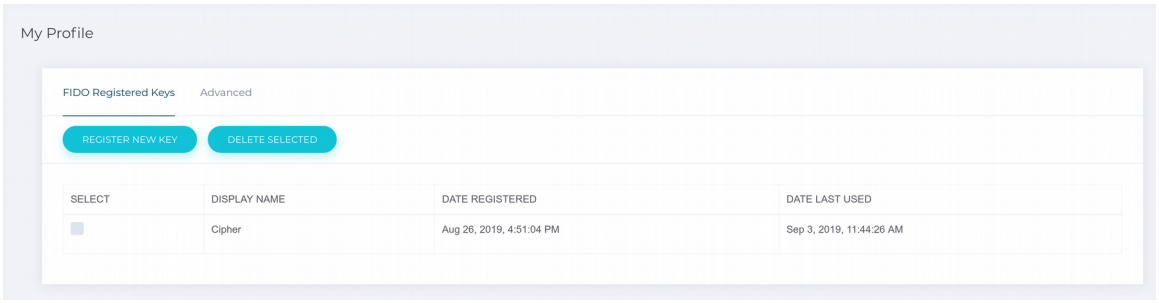


8. The browser may prompt using its default prompts. The first one shown here will close on its own once a FIDO device has been used to verify user presence. If the second one shown here appears, click **Allow** to continue; clicking *Block* will restart the registration process.



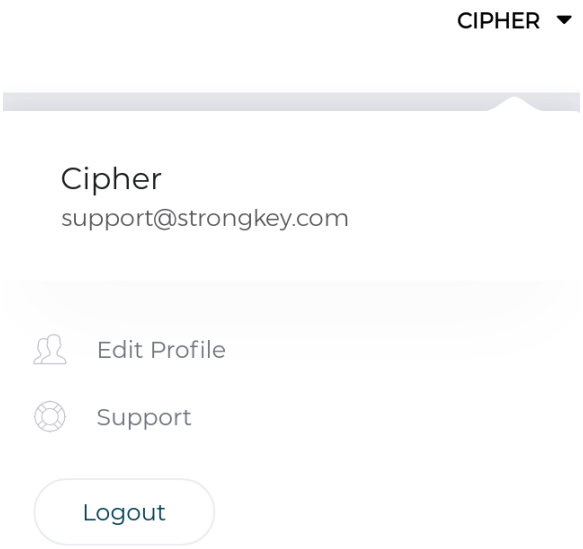


9. *My Profile* displays for the currently logged-in account, listing all FIDO devices associated with the account.



7.5—Logging Out

1. In the upper right of the page, open the *Account Menu* by clicking the **Account name**.

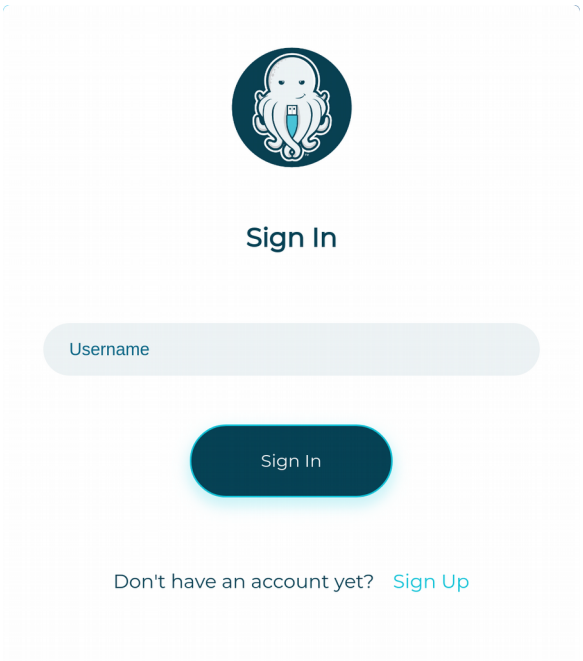


2. Click **Logout**. The default login page displays.

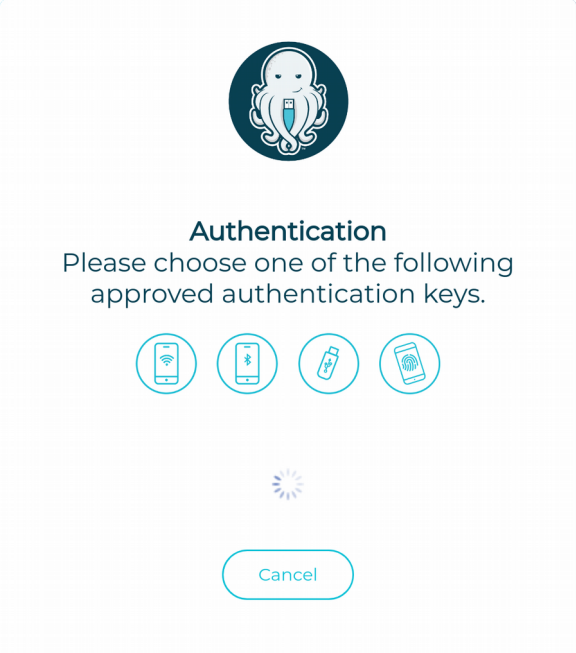
7.6—Authentication/Logging In

Now that SKFS knows you, you can authenticate without a password.

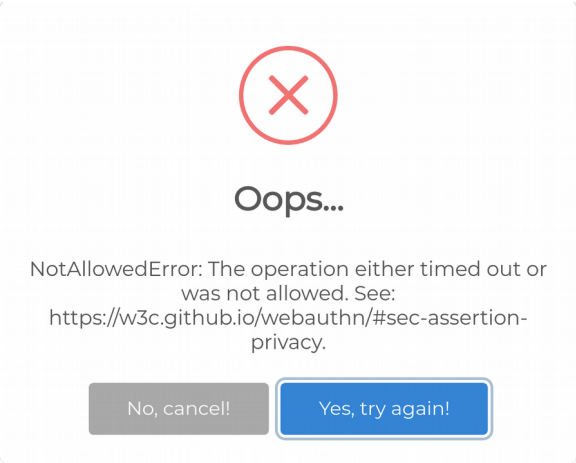
1. From a logged-out state, **browse to the FIDO2 Server**. The login page appears.

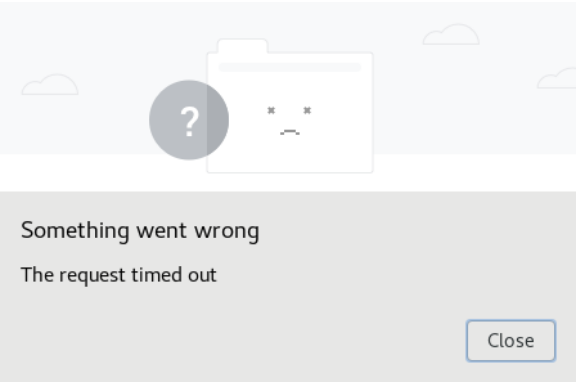


2. Type the login name and click **Sign In**.
3. You are prompted to **insert your FIDO device and touch it** to verify user presence.



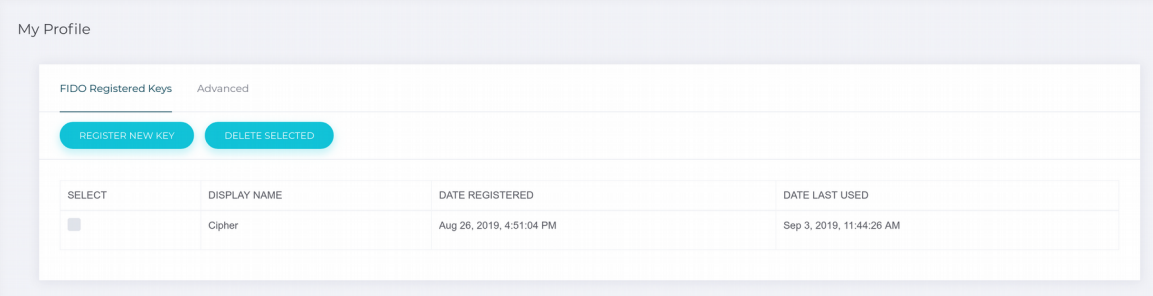
If the FIDO device is not plugged in and touched within a few minutes, the same screen displays asking if you would like to try again. A browser dialog may also display; if so, click **Close** to continue.





Click **Yes, try again!** to verify user presence again and continue to the *My Profile* page. Clicking *No, cancel!* returns to the login panel.

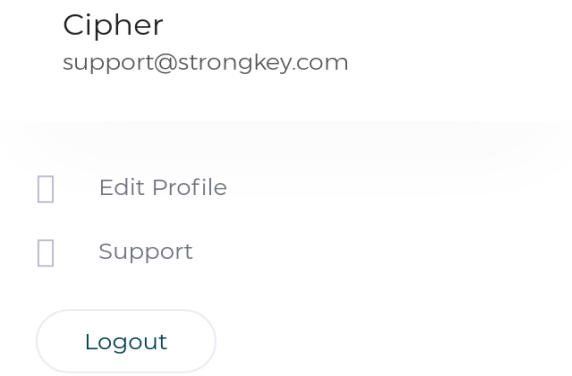
4. *My Profile* displays for the currently logged-in account.



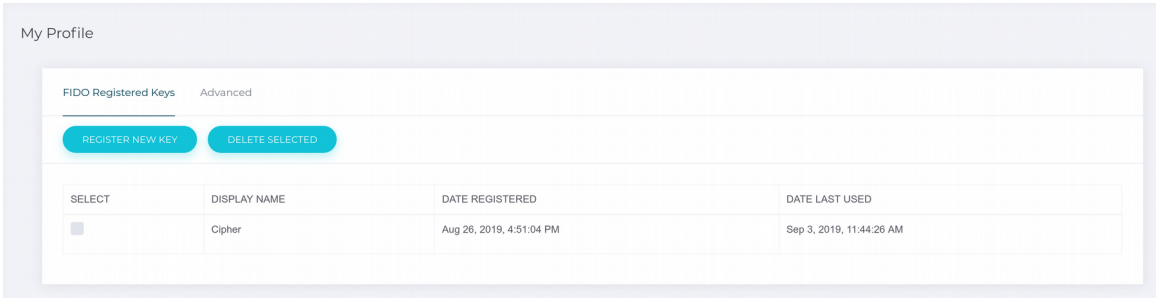
7.7—My Profile

Here you can view your account details, add/remove FIDO devices, and delete your account.

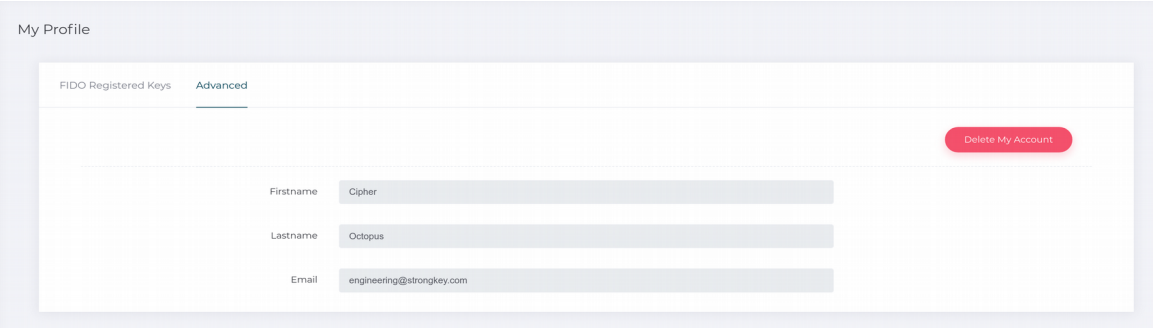
1. Make sure you are **logged in** to SKFS.
2. In the upper right, open the *Account Menu* by clicking the **Account** name.



3. Click **Edit Profile**.
   - The *FIDO Registered Keys* tab lists each FIDO Key with its label and its first and last use. The FIDO device used to register will initially be the only one listed. Each new device must be touched to verify user presence before the associated key will appear in the list.



- The *Advanced* tab allows deletion of the currently logged-in account and all associated keys by clicking **Delete My Account**. A prompt will ask for confirmation which, if accepted, logs you out and returns the application to the login page.



Appendix A—Key Management in the Cloud

A.1—Abstract

The cloud delivers benefits to businesses that simply cannot be ignored. However, with increasingly stringent security regulations, protecting sensitive data with encryption is mandatory in some segments. This leads to the question: where should cryptographic key management services be performed? This paper argues that, even when secure cryptographic hardware modules are used, there are residual risks with performing key management in a public cloud environment that can compromise secret keys and, consequently, the data they protect.

The paper also presents a hypothetical web application with a need to protect sensitive data through encryption, and describes multiple design paradigms for managing key management. It then describes risks in the Cloud that can compromise secret keys and the information they protect.

Finally, it points to a web application architecture that presents a solution to the problem where the benefits of the Cloud can be availed without compromising sensitive data or cryptographic keys that protect them.

A.2—Introduction

The public cloud represents a paradigm shift in how information technology is delivered to business units. Business problems are addressed faster because applications are delivered faster; labor costs are reduced because fewer technical people are required to manage larger infrastructure; change management becomes more responsive because operational tasks get accomplished faster; and businesses save time and money with “just-in-time IT” while being able to scale immensely if and when needed.

Since many success stories abound about the use of the cloud, the natural questions that follow from security professionals are: What about key management? Can key management not be performed in the cloud to deliver similar benefits to businesses as applications do?

Some *cloud service providers (CSP)* and *software-as-a-aervice (SaaS)* companies have offerings ranging from automatic file encryption in the cloud[i], to the use of multi-tenant cryptographic *hardware security modules (HSM)* in the cloud[ii]. The message to the market from the CSPs—and the market's perception—is that cryptography in the cloud is not only feasible, but can keep your data and cryptographic keys secure enough to comply with industry regulations.

Would that the answer were so simple.

The reality is complex because of a number of factors:

1. Cryptography—which can be used to preserve the authenticity, confidentiality, and integrity of sensitive-data—is the last bastion of defense from a compromise to systems that may result in a data breach. As such, cryptographic keys cannot be treated as other forms of data within a system.
2. Regulatory requirements around the world are becoming increasingly forceful about protecting sensitive data, and stipulating that "data custodians" prove they are in control of cryptographic keys when they encrypt data as part of their risk management strategy. Delivering adequate proof may be one of the most difficult aspects of cryptographic key management in the cloud.
3. The nature of the cloud encourages businesses to seek the most economical CSP to host their applications. Security, by its nature—and now, because of the law—is sufficiently complex that a detailed analysis of a CSP's security capabilities becomes counterproductive to the economics of using the cloud.
4. Distributed applications and evolving technology trends enable many design patterns for how applications are built and deployed. When applications must protect sensitive data, the security burden adds new complexity to applications whose architecture was sufficiently complex already.
5. The field of cryptography moves glacially. New algorithms or methods published in research papers must be first studied theoretically to determine if there are flaws in them that might reveal secrets. They must then be implemented into experimental software and subjected to attack through known tools and techniques. After many iterations, should anything hold up to scrutiny, many companies generally wait for "approval" by national standards bodies before adopting commercial versions of the software that must, typically, be certified by independent testing laboratories. This cautionary approach is intended to ensure safety in the adoption of new cryptographic algorithms and techniques even though it does not lend itself well to rapid advancement and adoption.

This paper only addresses risks with currently approved cryptographic technology without discussing experimental techniques—with the exception of highlighting a facet of one cryptographic technique that garners the attention of media from time to time.

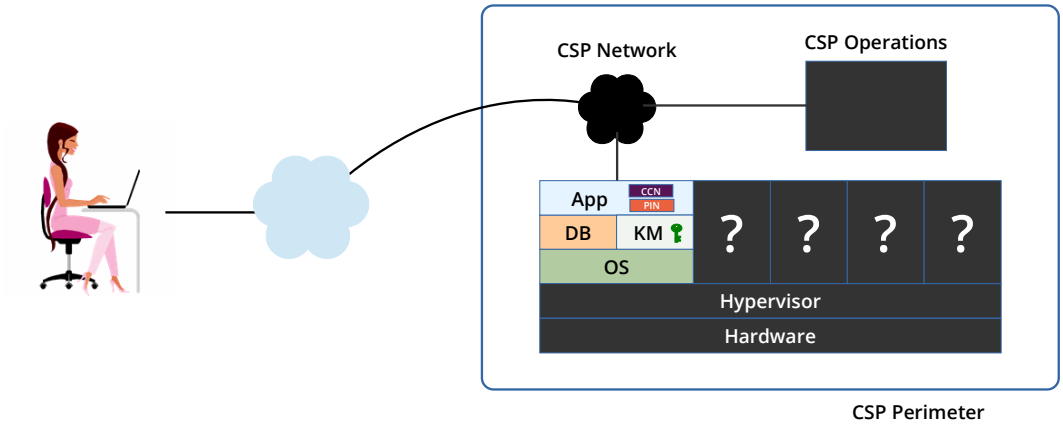
A.3—Background

To frame the issue, it is assumed companies are using the most basic and cost-effective cloud capability: *infrastructure-as-a-service (IaaS)*[iii]. While similar issues arise when using *platform-as-a-service (PaaS)*, the lack of true visibility in the implementations of popular SaaS applications does not provide sufficient information to determine their risk—however, there is evidence that SaaS applications are not immune from the problems identified in this paper[iv].

IaaS presents potential for the fastest migration from on-premises infrastructure to the public cloud as it enables the customer to duplicate application infrastructure operating within their own data centers or within managed service data centers, to almost identical infrastructure in one or more *Virtual Machines (VM)*.

PaaS solutions will almost always require the customer to modify their application to use the cloud service, while SaaS solutions will definitely require completely migrating from the customer's application to one provided by the SaaS provider, possibly entailing even modifying the business process a little. Both PaaS and SaaS will mandate the use of tools and components provided by CSPs. While some CSPs have started providing *key management as a service (KMaaS)*, this paper will show that customers are still exposed to risks despite these offerings.

This illustration represents a simplistic web application design as the "basis" for our discussion.



Assumptions we make about this design are as follows:

1. All components—web tier, application server, database, key management, etc.—are shown operating inside the same VM. This is merely for illustration; they could have been deployed within individual VMs and operating separately; but this paper will show that the security issues don't change in spite of the separation.
2. The VM used by the customer is hosted on a multi-tenant machine managed by the CSP's hypervisor, co-resident with VMs owned by other customers of the CSP.
3. The customer has no visibility into the physical hardware or hypervisor provided by the CSP. The CSP may be using any class of hardware or hypervisor that enables them to offer cloud services with the most competitive set of features and pricing.
4. Even if the CSP's network and operations are certified to an industry regulation such as the *Payment Card Industry Data Security Standard*[v] *(PCI DSS)* or equivalent, the customer typically has no visibility into the CSP network or operations save for a *service level agreement (SLA)* which the customer might expect and to which the CSP is held accountable.
5. The application may be using standard components such as Linux, Tomcat application server, MariaDB relational database, etc., and operates on sensitive data requiring data protection, such as financial data, *personally identifiable information (PII)*, healthcare data, critical infrastructure telemetry, etc.

6. The application encrypts data on the network and in the database using US *National Institute of Standards and Technology (NIST)* approved cryptographic algorithms from the Suite-B[vi] specifications. No assumptions are made whether the application chooses to encrypt data using operating-system-based encryption, disk-based encryption, file-based encryption, database encryption, or application-level encryption—this paper will show that they all present the same risks to the customer.
7. In this simple scenario, this paper assumes that cryptographic hardware modules are not in use and that secret keys are protected using *Password-based Key Derivation Functions (PBKDF)*[vii] or equivalent techniques for key protection.

> **NOTE:** The paper builds up to more complex key management models, but to establish a baseline the most common key protection mechanism is assumed.

A.4—Basis Application Issues

Performance issues aside, application users interacting with the basis web application are unlikely to see any functional difference in the application regardless of whether it is hosted on-premises or in the cloud. In fact, it is conceivable that users might even see better availability and performance given how rapidly a customer can scale the cloud application.

While administrators of the application are likely to see some differences when they deploy applications in the cloud versus an on-premises infrastructure, once the application is operating in a VM, even administrators are unlikely to see any difference. These are just some of the reasons that business owners are enamored of the cloud.

The problems with this picture are many:

1. The first has to do with the fact that customers typically want systems that stop for maintenance or other reasons to resume services as quickly as possible and with as little human intervention as possible. This typically requires that systems are designed in a manner where all components of the application system restart automatically. While most components in our basis application can restart without human intervention, the PBKDF-based cryptographic key will require human intervention to make the key available in plaintext (unencrypted) form to the system. It is conceivable that some customers might store the password to the cryptographic key in an application configuration property, to be automatically passed to the PBKDF at runtime to minimize restart latency. As one might note, having the password to the cryptographic key within the application—whether in a VM or not—is an insecure practice. In the public cloud, it is particularly dangerous for the reason noted next.
2. In the public cloud, multi-tenant machines and their VMs are managed by the CSP. Users—authorized or unauthorized—with administrative credentials to the hypervisor have the ability to:

   a. Copy a running VM and its contents to another physical server while keeping all the VM's contents intact[viii].
   b. Save an image of a running VM to a disk to restore onto another server with all the contents of the source VM.

Both operations can be carried out unbeknownst to the customer. Even if the customer does not use PBKDF in the cloud and injects keys into the application at runtime from outside the VM, once the plaintext key is in the VM, both administrative operations described here carry the cryptographic key from source to destination.

3. Administrative credentials for applications or the cloud infrastructure are typically based on shared secrets. Whether they are passwords based on secure message digests (hashes), *hashed message authentication codes (HMAC)* or *one-time passcodes (OTP)*, they manifest the same problem: shared secrets. The shared secrets are capable of being attacked at both ends of the transaction. While this problem is not unique to public clouds, that the cloud is a multi-tenant environment with the potential for significantly scalable attacks, it represents an inviting target to attackers.

   NOTE: Since the original writing of this paper, this potential has even led to an anonymous business venture to enable searching from billions of files – either improperly secured or simply not secured—within public clouds[ix]. Without strong-authentication leveraging public-key cryptography to administrative interfaces of the public cloud, it is impossible for anyone to prove a shared secret credential used to access sensitive information in the VM was from an authorized source.

4. A USENIX peer-reviewed paper[x] documents how a malicious VM can gain unauthorized access to a co-hosted VM running on the same hypervisor. Using the technique described in the paper an attacker can effectively steal cryptographic keys of another machine in the same cloud environment. As the authors of the paper state, "…existing cryptographic software is wholly unequipped to counter [the attack], given that bit flipping is not part of their threat model."

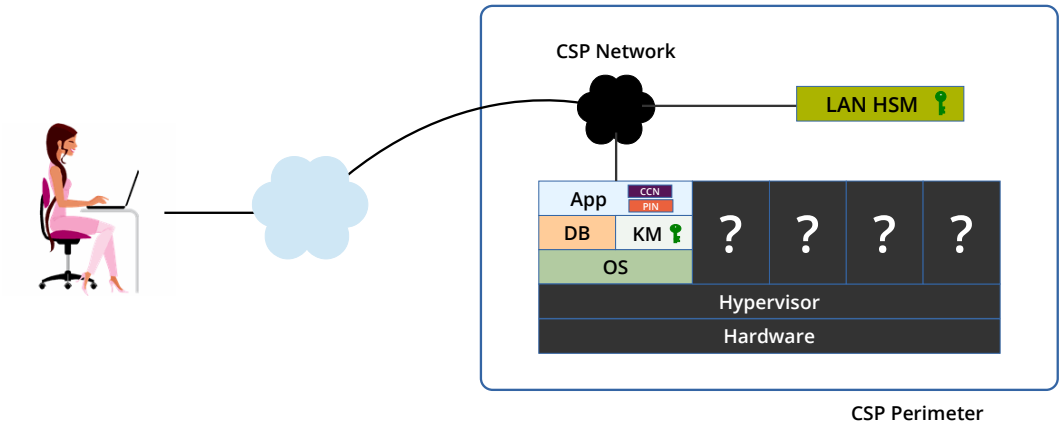
These are some of the possible attacks on applications that do not use any cryptographic hardware to protect and/or manage cryptographic keys. The next section describes potential vulnerabilities to sensitive data even when secure cryptographic hardware are used to protect cryptographic keys.

A.5—Secure Element in the Cloud

A standard practice in the field of cryptographic key management is to protect secret keys within secure elements such as the *Hardware Security Module (HSM)*, the *Trusted Platform Module (TPM)* or similar devices. HSMs and TPMs are highly specialized, purpose-built devices whose sole purpose is to generate, protect and manage the lifecycle of cryptographic keys, while providing applications access to the use of the keys. Designed with custom hardware, operating systems, and application programming interfaces, suppliers provide industry standard cryptographic algorithms for use by application developers to secure sensitive business information. Such devices have been in use for decades, primarily in the banking industry and military.

The theory for the creation of these specialized devices is, that even if the host computer and application are compromised, the cryptographic keys protected by them are safe since they're specially designed to protect against targeted attacks to cryptographic key material; there are even certification standards and laboratories established around the world to enable such devices to come to market using such standards[xi].

As with many things in technology, there many ways to implement the use of TPMs or HSMs with applications—one such implementation is shown in Illustration 2.



In this scenario, the CSP offers a key management service based on a slot of a *local area network (LAN)* attached HSM. The slot, usually based on the PKCS #11[xii] specification, enables a customer to manage the roles of Security Officer and User, while the CSP only retains the role of HSM Administrator without access to the HSM slots or the cryptographic keys within them (even though the CSP physically controls the HSM on their premises).

The benefits promoted for this design are:

1. The cryptographic keys necessary for operations over sensitive data are in the LAN HSM slot and not in the VM; the keys may be generated inside the HSM and never leave the HSM unless protected by a key-encrypting key through "key wrapping." Data to be protected is sent from the application executing within the VM to the LAN HSM for cryptographic processing. As a result, the secret key does not leave the LAN HSM, thereby mitigating the risk of cryptographic keys being exposed in the VM.
2. Just as a VM in a multi-tenant cloud offers business advantages for computing and storage, a slot in a multi-tenant LAN HSM offers similar business advantages for key management.
3. Unlike the control that CSPs have over VMs, CSPs do not have similar controls over the slot in a LAN HSM. The HSM Administrator may change configuration parameters on the device, create and delete slots, and perform maintenance tasks on the LAN HSM, but they neither have the privilege to access cryptographic keys nor create application credentials to do so. As a result, the customer's Security Officer has the sole authority to designate which application user may access secret keys to perform cryptographic operations.

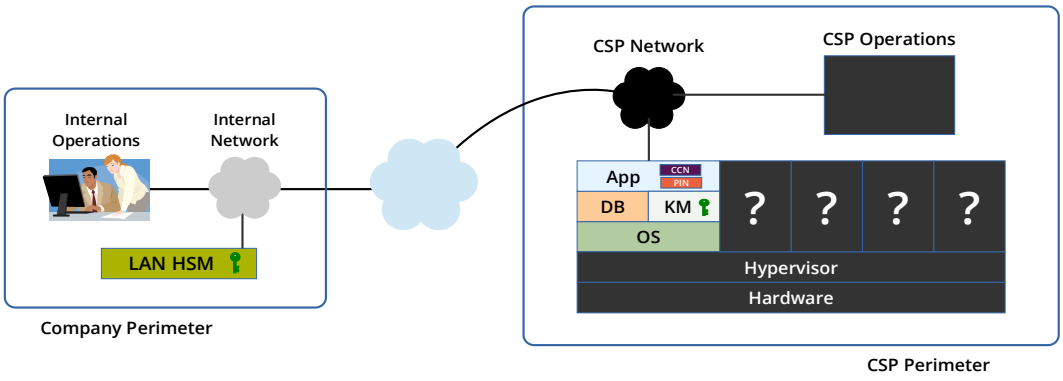
A.5.1—Secure Element in the Cloud Issues

While the security benefits of secure elements such as the HSM and TPM cannot be disputed, there are two issues with this design that CSPs do not necessarily highlight to prospective customers, unless pressed:

1. To use the secret keys in the LAN HSM slot, the application must have an authentication credential, typically called a service credential, to authenticate the application to the HSM. This always requires the use of a secret in one of three forms: a password, an HMAC key, or a private key of a public/private key pair. When the secret is used by the application, it must be exposed to the application in the VM, thereby exposing the secret to all the risks described in the previous section. Consequently, even though the keys in the LAN HSM themselves are safe, the credentials to access them from a VM in the cloud are not. An attacker is unlikely to care very much about attacking the LAN HSM directly if they can compromise the credentials to the LAN HSM and discover all sensitive data as it passes through the compromised VM. While this level of detail is generally unavailable to the public in any such breach, to an experienced technologist, its not very difficult to deduce what happened in one of the largest breaches in a public cloud to a very large bank that was most likely using such HSMs to protect sensitive data[xiii].
2. The second attack does not compromise sensitive data, but creates a *denial-of-service (DoS)* attack on the HSM that can impact the business operations of the customer while casting aspersions on the integrity of the LAN HSM and/or its operations within the CSP's environment. The attack consists of compromising the credentials of the HSM Administrator and deleting the slot and all its contents, and/or compromising the credentials of the Security Officer of the slot and erasing the keys in the slot. Since the design of PKCS#11 factored in role-based access control, it takes care to ensure that only application service credentials can use the cryptographic keys even though they may not have access to the key itself, while administrative users can manage the device and/or the logical slots on the device without access to the keys.

A.6—Secure Element On-premises

This is a variation of the Secure Element in the cloud scenario and is shown in Illustration 3.



In this scenario, the recommendation is to deploy the LAN HSM within the controlled premises of the customer while the application is executing in the VM in the Cloud, and create a secure link between the application and the LAN HSM. Furthermore, the application is designed to send sensitive data outside the cloud (over the secure link) to the LAN HSM, have it encrypted and brought back into the cloud for storage. The process is repeated for decryption with the ciphertext (encrypted data) being sent to the LAN HSM for decryption and, generally, returned to the application for further processing.

The benefits promoted for this design are as follows:

1. The cryptographic keys necessary for operations over sensitive data are in the slot of the LAN HSM and not in the cloud VM. Data to be encrypted and decrypted is sent from the application in the VM to the LAN HSM for operation. As in the earlier scenario, the secret key does not leave the LAN HSM thereby mitigating the risk of exposing cryptographic keys in the cloud VM.
2. The LAN HSM is not only physically stored on customer premises, but is also managed by the customer's operations staff. This effectively takes all physical and logical control of the device and its contents away from the CSP and places it within control of the customer. The only downside for this scenario is that the customer must fully invest in the LAN HSMs, rather than just a slot, and manage it with their own operations team.

A.6.1—Secure Element On-premises Issue

While the customer does have physical and logical control of the LAN HSM and its contents in this scenario, it does not address the issue that the plaintext authentication secret to the LAN HSM must be accessible to the application in the cloud VM, thereby exposing the secret to all the risks described in the Basis Application Issues.

From a risk-management point of view, it is counterproductive to spend a significant amount of money to build and operate a LAN HSM device on premises, and then expose the credential with privileges to use the secret keys in a public cloud VM. As was documented earlier, attackers *don't mind* having just the credentials to use the cryptographic keys in the LAN HSM than to have to attack the LAN HSM directly.

A.7—What about Public Keys in the Cloud?

StrongKey believes the industry is on the threshold of adopting new authentication standards from the FIDO Alliance[xiv]. Given this, a question arises: Since the FIDO protocols only store raw public keys and user key handles—opaque BLOBs defined in the FIDO protocols that may contain encrypted private keys—is it safe to store and rely upon FIDO public keys when strongly authenticating users to web applications in the cloud?

On the surface, it would appear that storing and relying upon FIDO public keys in the Cloud should be safe since public keys, by definition, may be disclosed in public. Indeed, they are, in most use cases. However, it is this author's position that the cloud is different—a and this changes what we have always known to be generally accepted practice about public keys.

A.8—Substitution of Keys Attack

This attack works because the privacy requirements in the FIDO protocols prohibit references to the user's identity within the public key; and since the key handle is opaque to the FIDO server, the FIDO authenticator implementer is free to define it as they see fit. To ensure interoperability of FIDO authenticators, the FIDO server is neither required to peek into the key handle nor understand its data structure, but to dutifully store it during FIDO registration[xv], then to retrieve and send it back to the user during FIDO authentication[xvi]. The binding between the user and her public key is in the metadata of the datastore or database of the FIDO server. This is why a simple substitution of public keys and key handles enables the attacker to compromise other accounts.

The standard practice to mitigate such attacks—not just for FIDO protocols, but also within any business application where the veracity of its data is critical to its business operations—is to verify the integrity of the user registration object/record on every read operation in the datastore. This is accomplished by applying a digital signature to the object/record upon persistence and verifying the signature upon reading it back to ensure the object/record has not been modified. This practice, however, requires that the FIDO server have access to a signing key (a.k.a. a private key) to create the digital signature at persistence time. Since this introduces a secret that must be protected, the issues described earlier in this paper come back to haunt even a datastore of mere public keys.

A.9—Summary of Issues

For the key management design paradigms described here—a few of which even use HSMs to protect secrets—given the nature of the public cloud and the documented breaches in the cloud, data custodians will have a difficult time proving that they have full control of cryptographic keys—or access to them—from a cloud VM. While they can certainly show they have stipulated and followed industry best practices in managing cryptographic modules and keys, the public cloud makes it impossible to prove that someone unauthorized does not have access to the secret keys in the cloud. This vulnerability gap is simply because the public cloud was never part of the threat vector for cryptographic key management systems.

Is there any technique or cryptographic algorithm that permits the use of encrypted data to make business decisions without the need for secret keys to decrypt such data?

A.10—Homomorphic Encryption

A technique, called the homomorphic encryption scheme, presented in a dissertation paper[xvii] shows how one may compute arbitrary functions over encrypted data without the decryption key, thus enabling searches and computations on encrypted data.

It would appear that homomorphic encryption is the perfect solution for the public cloud. However, it is important to understand that computations using the homomorphic encryption scheme assume you are working with encrypted data. What the homomorphic scheme does not specify is where the encryption must occur before the computations are performed. Nor, does it specify where decryption of encrypted data must take place to validate the answer to the homomorphic computation. To some, it might seem obvious that encryption and decryption occur where data is stored. However, this paper has identified the risks of performing cryptographic operations in a VM in the public cloud. To be truly secure with the use of homomorphic encryption, sensitive data must be encrypted outside the public cloud before encrypted data can be used by applications designed to work with homomorphic algorithms in the cloud. That defeats the purpose of the homomorphic scheme, since all the risks identified in this paper are back on the table.

What, then, is the solution to this conundrum?

A.11—Regulatory Compliant Cloud Computing

This author has published a paper[xviii] describing a web application architecture to address this problem.


NOTE: This architecture is documented in *Appendix B*.

It is the author's contention that paradigm shifts in technology sometimes mandate similar shifts in application architecture. The software industry had to do this in the 1980s when monolithic applications on mainframes and minicomputers were migrated to personal computers and LAN/UNIX servers: the client-server architecture was necessary to take advantage of the technological and business benefits that the change represented.

The paradigm shift occurred once again during the 1990s when the world-wide web came to be: three-tiered web application architecture took root, and continues to remain a staple of web and mobile applications even today.

The changes represented by the cloud mandate a similar rethinking in how applications are designed and deployed to take full advantage of the benefits of cloud computing. The mandate for security is, coincidentally, driving national strategies[xix] for how applications should be designed to comply with such regulations[xx].

This author proposes that applications taking advantage of the cloud and with the need to be secure must consider modifying their application architecture to take advantage of the *Regulatory Compliant Cloud Computing (RC3)* architecture. While the referenced article provides more details, very simply it consists of the following:

1. Classifying information into three categories:
   a. That which must be protected by law.
   b. That which must be protected by company policy.
   c. All other information deemed public information.
2. Creating two zones of computing:
   a. A secure zone to handle sensitive information.
   b. A public zone to process all non-sensitive information computing and storage.
3. Information is protected in the secure zone and may be stored in the public zone, but must be brought back to the secure zone for processing.

   Communications between the secure zone and the public zone are one-way: from the former to the latter. The public zone may never initiate a connection to the secure zone. The user, however, interacts with both zones depending on where the application transaction takes them.
4. Designing applications to ensure that all application transactions that deal with sensitive information are processed and protected only in the secure zone, while the public zone processes all non-sensitive information.

The RC3 architecture is simply an evolution of the computing models that have been in use over the last fifty years of software development, taking into account the cloud and the need for security to be ingrained within the application.

While there is a lot more to the RC3 model for application design, the model has been proven to work[xxi] and has the benefit of being able to take advantage of any cloud in the world without concern for the security controls provided by the CSP.

Until such time cryptographic techniques evolve to solving the need to balance cloud benefits with the security mandate in a simpler manner, RC3 presents a viable compromise to address the problem.

A.12—Summary

This paper identifies risks with cryptographic key management and storage of secrets in the public cloud that cannot be eliminated easily. Given the current state of the art, the logical solution is the modification of application architecture to process sensitive data—including cryptographic key management—in a secure enclave outside the public cloud while leveraging the public cloud for all non-sensitive data.

Appendix B—Web Application Architecture for Regulatory Compliant Cloud Computing (RC3)

B.1—Introduction

Unless your organization is unique, not all your data is sensitive. The logical approach should be to build your IT infrastructure to optimize your investments, protecting what matters while allowing non-sensitive data to run free.

This paper presents architecture for next-generation web applications. It leverages emerging technologies such as cloud computing, cloud storage, and *enterprise key management (EKM)* to lower costs, speed time-to-market, and scale with variable investments—while proving compliance to PCI DSS, HIPAA/HITECH, and similar data security regulations. We call this architecture Regulatory Compliant Cloud Computing, or RC3.

Cloud computing presents many opportunities as an alternative deployment strategy for IT systems, challenging traditional notions of data security. The increasing enforcement of data security regulations1[1] leaves information technology professionals perplexed on how to take advantage of cloud computing while proving compliance to regulations for protecting sensitive information.

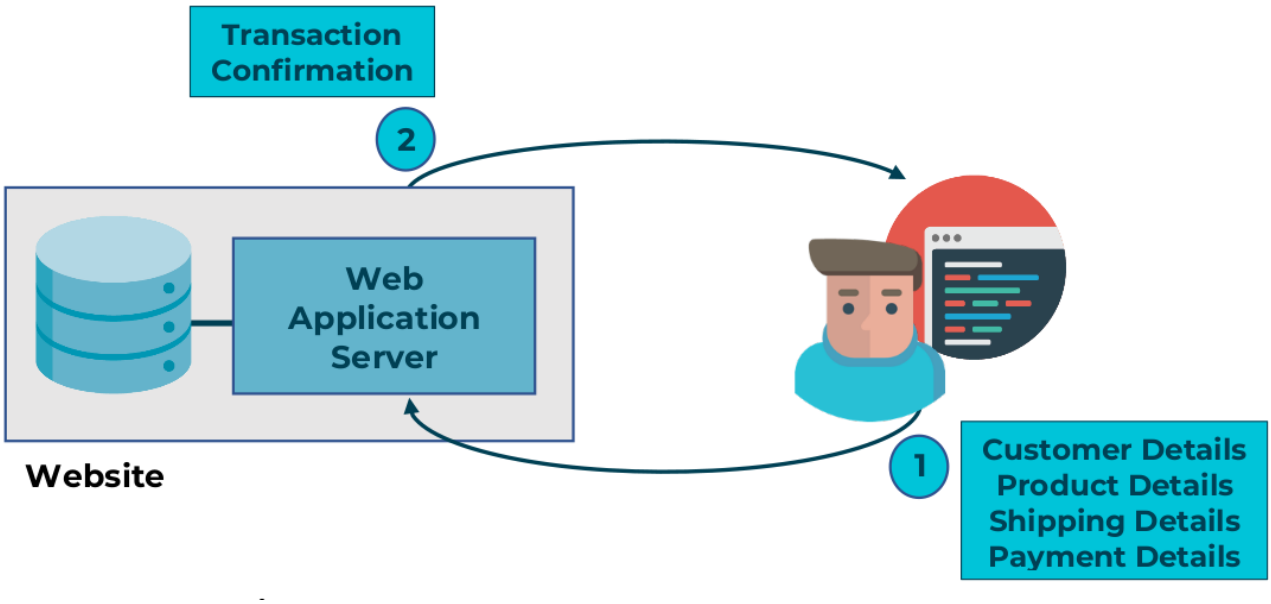
The dominant positions gravitate towards either not using the cloud at all or embracing it completely. We believe the optimal solution is in the middle: with sensitive data secured and managed within controlled zones, while non-sensitive data lives in clouds. This allows you to prove compliance to data security regulations while leveraging clouds—private or public—to the maximum extent possible.

This paper describes how a specific web application architecture optimizes IT investments with cloud computing, while remaining compliant with data security regulations.

1 TJX and Heartland Payment Systems together paid more than USD 220M as fines and settlements for their breaches of 94M and 130M credit card numbers in 2007 and 2009, respectively. These breaches represent the largest publicly disclosed breaches of sensitive data from computer systems.

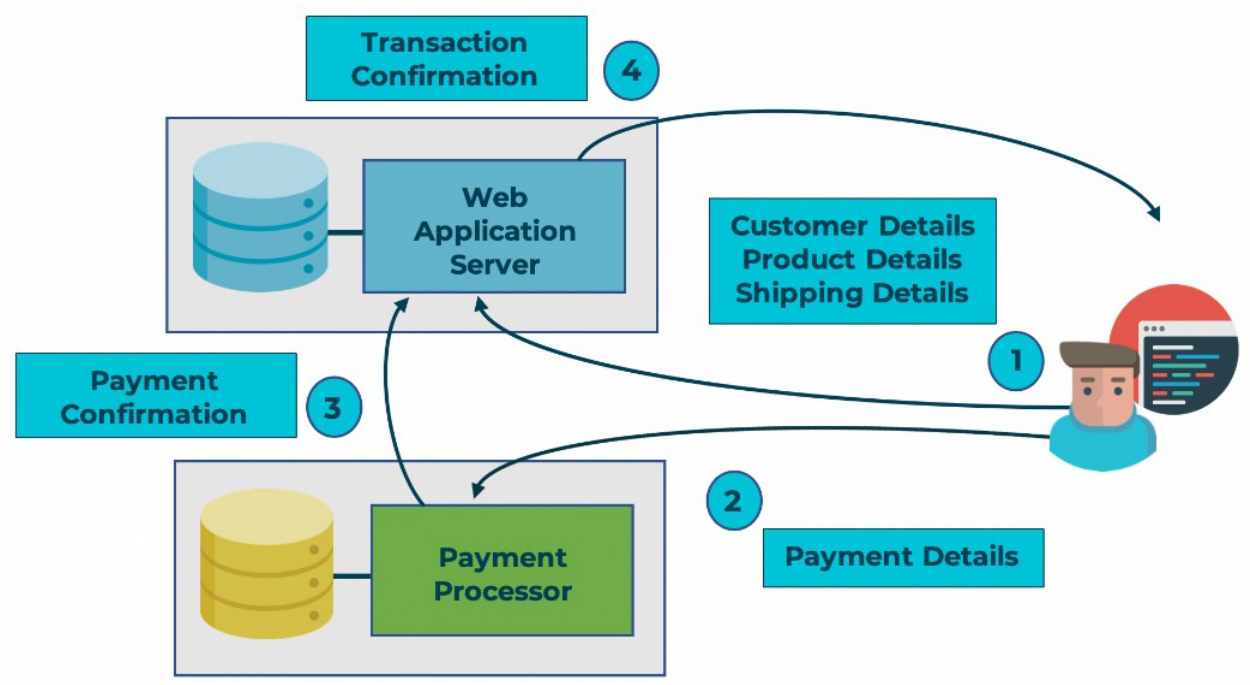
B.2—Current Web Application Architecture

Conceptually, web applications are simple. The browser (the client in a client-server connection) displays a form and requests data from the user. The server (the server in a client-server connection) is a program on a web server. The user submits the form, the server program receives and processes the information, then a response is returned based on the outcome. This interaction is shown here:



While the model can get complex, the common feature among them—the web form—must identify the server *Uniform Resource Locator (URL)* so the browser knows where to send the form data upon submission. Users are typically shielded from the complexities of redirection, allowing them to perceive the transaction as seamless.

In some cases, such as e-commerce applications, the browser is redirected to a payment processor site, then back to the original site to finish the transaction. The advantage for the e-commerce site is outsourced infrastructure for the payment processing part of the transaction. This redirection is shown in the image here:



B.2.1—Disadvantages of the Current Mode of IT Investments

Assuming a typical e-commerce application as an example, here is a list of responsibilities with the current mode of IT investments:

- You must procure physical resources—computer, storage, and network—for all functions of the application: customer registration, product management, inventory, purchase transactions, payment processing, fulfillment, and many others. This usually leads to the additional burden a few years later of managing the transition from the installed infrastructure, as components age and fall behind desired performance requirements, to newer infrastructure.
- You must ensure redundancy of the computing infrastructure for business continuity—usually doubling the infrastructure investment.
- You must secure the entire infrastructure. Since most sites do not distinguish between sensitive and non-sensitive data, the security framework usually applies to all components of the infrastructure and data[2]. This represents a misapplication of resources, since non-sensitive information does not need the same degree of protection as sensitive resources.

This mode of investing has not changed for the last 40 years. While the capital outlay

2 In the last few years, because of PCI DSS, sites make a distinction between a "PCI zone" and "non-PCI zone," "PCI data," and "non-PCI data." As such, the "PCI zone" and "PCI data" typically receive more attention and investment from a security standpoint than the non-PCI zone or data. While this might be considered as a form of optimization, because the non-PCI zone is still within the network perimeter of the site, you are still spending more than what you might if the application were redesigned with the new web architecture described in this paper.

has come down dramatically from the days of the mainframe, an application that must serve hundreds of thousands of users still requires a sizable capital outlay despite availability of commodity servers and open-source software.

B.2.2—Cloud Computing

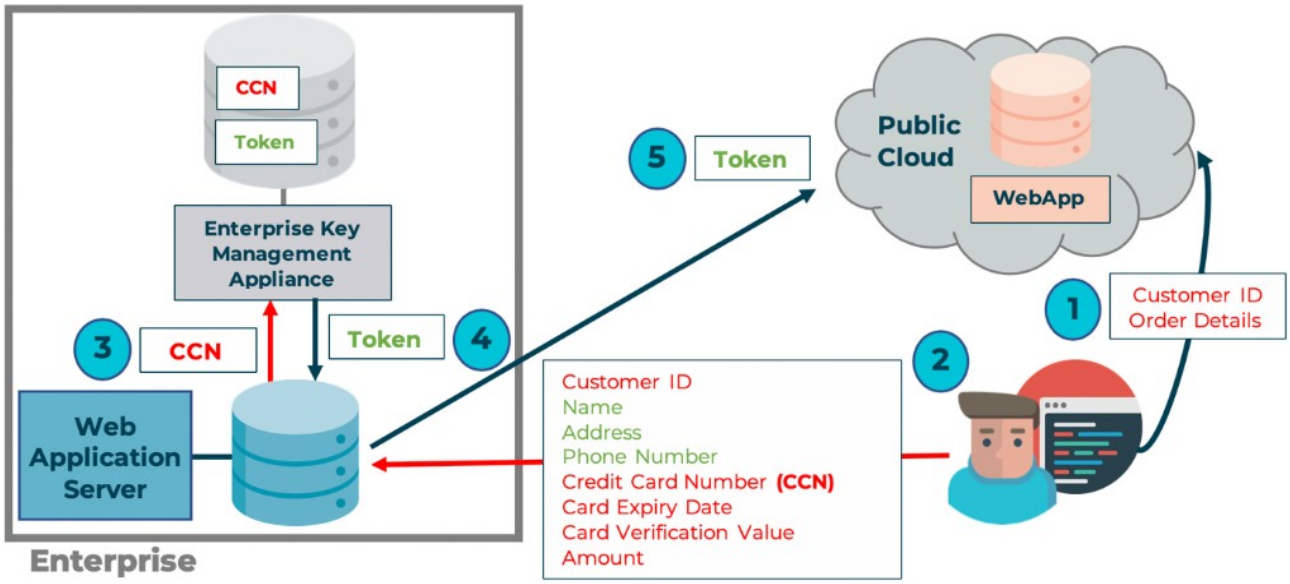
The emergence of cloud computing technology—especially public clouds—dramatically changes how such IT investments can be made. It is no longer necessary to make large, risky investments up front and depreciate those investments over the course of many years. With much smaller outlays, companies can design exactly the IT services they need and pay for only what they use. The economic impact of this change cannot be overstated as new businesses come to market on significantly smaller budgets.

As significant as this change will be on delivering and managing IT services, the burden of securing sensitive data cannot be outsourced. While it may be contractually delegated to a third party, the responsibility of ensuring compliance to security regulations still remains with the owner of the data. As such, we believe architects and web application designers will find the model described herein useful in meeting their compliance obligations while taking full advantage of cloud computing.

B.3—Regulatory Compliant Cloud Computing (RC3)

Business transactions consist of a mix of sensitive and non-sensitive data. What is deemed sensitive, and the percentage of sensitive to non-sensitive data, varies depending on the business and the type of transaction. For the vast majority of businesses, assuming a normal distribution, the ratio of non-sensitive to sensitive data will be 4:1. Given this, the efficiency of IT investments can be improved by computing, storing, and managing sensitive data within regulated zones inside a secure perimeter, while non-sensitive data can be computed, stored, and managed in public clouds.

RC3 is the term given to the model of computing where business transactions span regulated zones with public clouds. Sensitive data is encrypted, tokenized, and managed in the regulated zone within the secure perimeter of an enterprise (or a delegated outsourcing company), while all non-sensitive data resides in the public cloud. This is shown in the image here:



B.3.1—Data Classification for RC3

A prerequisite for building RC3 applications is to classify data into three categories. This is necessary so applications can be designed to deal with data accordingly, and to simplify communication between business units and technical people who develop and support IT services:

#	Description
1	**Sensitive and Regulated Data** Data whose disclosure to the public would result in fines, potential lawsuits, and loss of goodwill to the breached entity. Examples are: credit card numbers, social security numbers, bank account numbers, etc. This type of data is designated as **Class-1** or **C1** data in this paper.
2	**Sensitive but Non-Regulated Data** Data whose disclosure to the public would result in embarrassment and some loss of goodwill to the breached entity. Examples are: an employee's salary, sales figures for specific product lines, name, gender, and age of a customer, etc. This type of data is designated as **Class-2** or **C2** data in this paper.
3	**Non-Sensitive Data** Data whose disclosure to the public would be insignificant to the breached entity. Examples are: product descriptions, images, etc. This type of data is designated as **Class-3** or **C3** data in this paper. It should be noted, that when sensitive data is tokenized in a well-designed *encryption and key management (EKM)* system, it is effectively rendered non-sensitive. Thus, even C1/C2 data can be classified as C3 after it has undergone encryption and tokenization.

Based on the above classification, companies adopting RC3 will ensure the following:

- All *C1* data will be processed and stored in regulated zones, within a secure network perimeter; these zones will prove they are compliant with applicable data security regulations; *C1* data tokens—sensitive data that has been encrypted and replaced with tokens—may be stored in public clouds
- All *C2* data will be processed in secure, but not necessarily regulated, zones. *C2* data tokens may be stored in public clouds
- All *C3* data may be processed and stored in public clouds

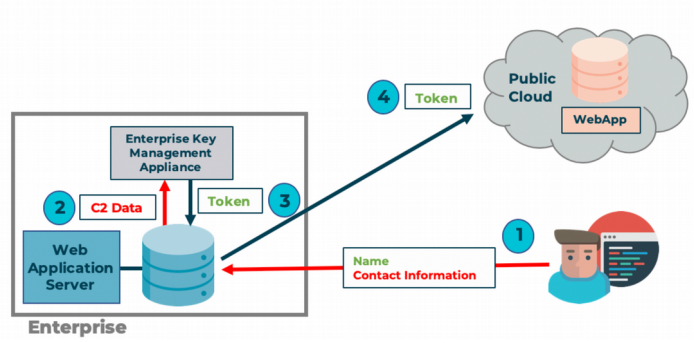
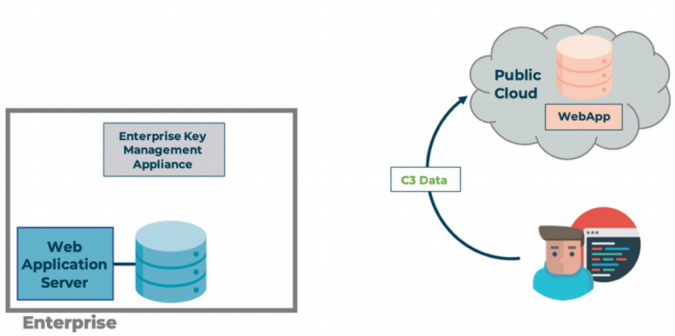
Applications must be written to deal with this separation of data; but the web application architecture—specifically, the ability to redirect the browser to targeted servers—lends itself to support this model. The next section describes a few examples of transactions in different sectors. The model, however, can be applied to any industry that faces similar challenges.

B.3.2—An E-commerce RC3 Transaction

This example, depicted at a high level, is described using the Java application model. However, the model is not exclusive to Java and can be easily duplicated in the .NET framework, or using other scripting languages such as PHP, Ruby, etc. Additionally, while the examples might show the use of the *Amazon Web Services (AWS)*, this is merely for illustration; the model is easily duplicated in any of the other public cloud infrastructures such as Azure, vCloud, etc. Finally, the examples show the use of StrongKey's CryptoCabinet and KeyAppliance in the regulated zones to secure C1 and C2 data.

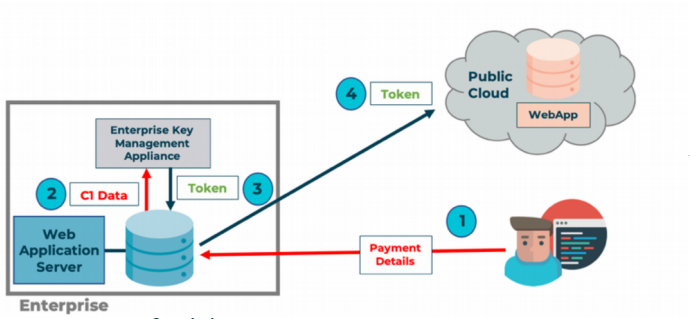
The regulated zone consists of a company *demilitarized zone (DMZ)* and a *secure zone (SECZ)*. A web application server resides in the DMZ receiving connections from users on the internet. It communicates with a database server, a CryptoCabinet, and a KeyAppliance residing in the SECZ. All communications are over TLS/SSL.

The *public cloud zone (PBZ)* consists of a web server and a data store. The web server receives connections from users on the internet, as well as web service requests from the web server in the company DMZ. All communications are over TLS/SSL. Web service requests from the company DMZ to the public cloud are further secured by SSL ClientAuth for mutual authentication between endpoints.

#	Description
1	The User registers themselves as a customer in the regulated zone, and is assigned a unique *Customer ID (CID)* which is treated as C3 data. The customer name and e-mail address is designated as C2 data while the customer's street address is designated as C3 data. C2 data is encrypted, tokenized, and stored in the KeyAppliance. All C3 data is stored in the PBZ and transmitted over the client-authenticated SSL link along with session-related data for this transaction. 
2	The user's browser is redirected to the PBZ at this point, where it processes most of his transaction: • Reviewing a list of products • Determining their price and availability • Adding selected products to the cart • Providing shipping instructions • Any other non-payment0-elated data The request headers carry session tokens from the web server in the DMZ; this allows transaction data in the PBZ to be correlated to the same transaction in the regulated zone. 

3 When ready to checkout, the user's browser is redirected to the company

Upon confirming the transaction, the sensitive C1 data is encrypted, tokenized, and stored in the KeyAppliance. Once tokenized, the C3



Some security notes about the e-commer

1. Compliance to data security regulatio regulated data is encrypted and store control of a cryptographic hardware module managed by three key custodians. The cryptographic keys that encrypt the sensitive data never leave the KeyAppliance and will decrypt the tokens to authorized users, but only inside the appliance.
2. The PBZ does not store any credential information for the user. User authentication is performed in the regulated zone, a valid session token is assigned to this user and the user's browser is redirected to the PBZ for further processing.
3. Communications between the DMZ and PBZ are only from the DMZ to the PBZ. The PBZ **never** communicates with servers in the regulated zone—if the application is designed appropriately, there is no need to do so. This ensures that any compromise in the PBZ never spills over into the regulated zone.
4. Servers from the regulated zone communicate with the PBZ only over SSL client-authenticated web services. This avoids the need to store any authentication credentials in the PBZ[3].

3 SSL client authentication only requires the storage of a valid and trusted digital certificate on the target machine to authenticate a client connection. The client, however, must possess a valid private key to the digital certificate and participate in the SSL ClientAuth protocol.

B.3.3—A Healthcare RC3 Transaction

This example, depicted at a high level, is similar to the e-commerce transaction. However, this transaction goes further by showing how large blobs of unstructured data, such as an X-Ray image, can also be stored in the PBZ while proving compliance. It is assumed that basic information about the patient was already created prior to this transaction.

#	Description
1	A technician at an X-ray lab authenticates herself to servers in the regulated zone of a hospital and establishes a session. If new patient data needs to be created, this is done in the regulated zone where a *Patient ID (PID)* is assigned. Some elements of the patient's demographic data are designated as C1/C2 data; as such, this is encrypted and tokenized by the KeyAppliance. The hospital has the choice of keeping the tokenized C1/C2 data within the controlled zone, or it may also store them in the PBZ using the secure one-way web service into the cloud.
2	The technician's browser is redirected to the PBZ where she submits the non-sensitive part of the transaction, such as: • Date and time of the visit • Requesting doctor's identifier and his/her prescription for the test • Attending technician and actions carried out • Any other non-sensitive data The application is designed so that this part of the transaction does not carry any C1 or C2 data.
3	When ready to submit the X-ray image and the radiologist's report, the technician's browser is redirected to the regulated zone. The technician uploads the X-ray image and the report, which may be converted to an XML document by the web application. The rather large XML document consists of C1 data, which must be secured. The C1 data is received in the DMZ web server and sent to CryptoCabinet for encryption. A symmetric key is generated and used to encrypt the document contents. The symmetric key is escrowed on the KeyAppliance, while the encrypted X-ray image and report are stored in the PBZ through a secure web service request.

All security notes that apply to the e-commerce transaction also apply to the healthcare transaction. The only difference between the two transactions is the addition of unstructured data—the X-ray—to the healthcare transaction, thereby requiring the use of the CryptoCabinet for encryption of sensitive data. The KeyAppliance is designed to encrypt smaller and structured data, and is thus not appropriate for binary large objects such as images, audio, and/or video clips.

B.3.4—A Manufacturing RC3 Transaction

This example shows an engineer in an industrial setting, submitting a sensitive document such as a blueprint with a *bill of materials (BOM)* to an assembly line for manufacturing.

This type of transaction follows these steps:

#	Description
1	An engineer authenticates to servers in the regulated zone and establishes a session. The engineer is then redirected to the PBZ. A web service request securely transfers session-related information from the SECZ to the PBZ. In the PBZ, the engineer creates a new transaction that only accepts C3 data into the cloud. The transaction is assigned a unique transaction ID and returned to the browser of the engineer in the request's response headers. Since the transaction is for the creation of a new part by the manufacturing plant, the public part of the transactions accepts the non-sensitive components of the BOM.
2	The engineer's browser is redirected to the SECZ where the sensitive part of the transaction is submitted. This is information such as: ➢ The blueprint of the object to be manufactured ➢ The sensitive parts of the BOM ➢ Special instructions about the assembly, if any ➢ Any other sensitive data The application is designed so that this part of the transaction carries necessary C1 and C2 data for encryption and tokenization in the SECZ. The encrypted blueprint is saved in the PBZ since it is now desensitized.

All security notes that apply to the previous transactions apply to this transaction, too.

B.4—Conclusion

In summary, it is possible with an appropriate EKM to use public clouds for storing sensitive data while keeping compliance to data security regulations. The technologies to enable this is currently available; what remains is for application designs to take advantage of them—to design cloud applications so that they apply the appropriate level of security resources to differently classified levels of data the application accesses to function.

Appendix C—New Features

C.1—StrongKey Android Client Library (Preview Release)

C.1.1—Introduction

The *StrongKey Android Client Library (SACL)* is an open source, native Android library providing support for the FIDO2 protocol for native Android apps. It provides the following features:

- It is supported on Android 9 (API 28) "Pie" or greater
- It supports the Java programming language, and does *not* require the use of JavaScript or the WebView component to deliver FIDO capability—it does *not* support the use of external *Security Keys*
- It uses the *AndroidKeystore*—which takes advantage of the *Trusted Execution Environment (TEE)* or a *Secure Element (SE)*, if present—for key generation, storage, and usage. It is always used as a *user verifying platform authenticator (UVPA)*
- It supports *registration, authentication,* and *transaction authorization* using "dynamic linking"—a core requirement of the European Union's *Payment Services Directive, Revised (PSD2)* regulation for *Strong Customer Authentication (SCA)*
- It supports Android's *BiometricPrompt* API for verifying users before enabling use of the FIDO key (or alternate device authentication schemes—PIN, pattern—where biometric capability is unavailable);
- It has out-of-the-box integration with the open-source FIDO Certified® ***StrongKey FIDO Server (SKFS)***—just add your mobile app to the flow;
- It includes a sample e-commerce web application—the *Sample FIDO App for E-commerce (SFAECO)*—demonstrate four basic functions:
  - User enrollment
  - FIDO registration
  - FIDO authentication and
  - *Authorization* of business transactions with the registered FIDO key
- The server side components of the SFAECO app are installed on a demo
- It includes a sample browser based web-application to work in concert with the SFAECO app to perform sample back-office business functions. But, the primary purpose is to demonstrate the use of FIDO for strong authentication and to review business transactions performed by app users, as well as see data collected by the app when performing *transaction authorization (TXA)*;
- It includes a second sample browser based FIDO Key Management web-application to demonstrate the newly integrated *single sign-on (SSO)* capability of the SKFS with *JSON Web Token (JWT)* using x509 based *JSON Web Signatures (JWS)*;
- These web applications have been installed

SACL has been tested using Essential PH-1, Google Pixel 3a, and Google Pixel 4a phones—the first two running Android 9 (Pie) with API 28, and the Pixel 4a with Android R (API 30). The device must have a fingerprint enrolled to support the use of SACL. While it is likely to work on most Android devices with biometric capability, your mileage may vary.

This is a preview release; some things may be a little rough around the edges. However, it provides insight into the capability the completed product will have, and enables early adopter customers to design and code native Android business application(s) to work with the SACL. StrongKey's goal is to deliver a production-quality release by the end of June 2021.

Feedback on the SACL may be sent to getsecure@strongkey.com. Alternatively, post ideas on the forum of the repository where you downloaded the distribution. Thank you for using the SACL to secure your customers' and your data.

C.1.2—Notes and Known Issues

#	Issue
SACL-0001	In this Preview Release 1 (PR1), SACL is designed to allow multiple credentials to be registered from the SFAECO app, to the same *Relying Party ID (RPID)*. Normally, this would not be allowed, since an Authenticator designed for use by a single user must only have one credential registered to an RPID active at any time. However, this can be very cumbersome when designing your business app and testing it with one mobile device. As such, PR1 does not enforce uniqueness of FIDO keys to a specific RPID within the device. When StrongKey finalizes a Production release later this year, this enforcement will become default.
SACL-0002	On the Pixel 3a and 4a, WiFi services will not automatically turn on, resulting in error messages when attempting any function requiring access to the SFAECO services. Please check for these messages in Logcat if they don't show up in a Toast message. **Workaround:** If your phone is configured to work with your WiFi router, but fails to connect, it could be for a variety of reasons: Location Services are off; Data Saver is on; Advanced Network settings permit the phone to turn off network services when the device goes to sleep, etc. Use the browser on your phone to visit any website. This triggers network services to start. Once started, the SFAECO app will work fine. However, if the device goes to sleep even for a few seconds, it is likely to turn networks services off. You may have to repeat the process to get back on the WiFi network—or turn Location Services on to keep the service on.
SACL-0003	When a user enrolls, and if the device goes to sleep, the SACL will sometimes not persist the *counter* value for the Authenticator. This will result in error messages or crashes because the SKFS default security policy is to require that Authenticator counter values always be incremented. Logcat messages will show exceptions with an *HTTP 500* error. **Workaround:** Should this occur, **Force Stop** the app in your *Application Settings*. Make sure that your network is working (by using a browser to visit any website), and try it again. You should notice that the *counter* value should be greater than "1" in the *Registered Key* page.

#	Issue
SACL-0004	SACL has a default value of 5 minutes for enabling use of the *AndroidKeystore* after the user has successfully unlocked the mobile device using their PIN, pattern, or fingerprint. This is designed to simulate the *Regulatory Technical Specification (RTS*) of the EU PSD2 regulation. As a result, no biometric prompt will show up during FIDO key generation or during authentication with the FIDO key. However, a biometric prompt will *always* show up when the user is authorizing a payment transaction, displaying the "dynamic link" with appropriate payment information. The app was deliberately designed to show different modes in which a FIDO operation can work depending on how the app chooses to use the SACL—either by relying upon the default timeout for the use of *AndroidKeystore* on an unlocked device, or explicitly prompting for a user's fingerprint when confirming a transaction. If a FIDO registration or authentication operation fails in the application, this is most likely due to network timeouts; a secondary possibility is that it has been more than 5 minutes since the user unlocked the device. **Workaround:** Lock the phone by pressing the Power button, unlock it with PIN, pattern, or fingerprint; then restart the application to perform the necessary operation. It will succeed this time.
SACL-0005	The *Authenticator Attestation Globally Unique Identifier (AAGUID)* used by the Preview release of SACL is **CAFEBABECAFEBEEF0123456789ABCDEF**. When SKFS goes into *Generally Available (GA)* status as a production quality release, the AAGUID will become **5341434C323032304b4F52D999D03ECB**.
SACL-0006	Testing has not been performed on multiple apps using SACL on the same mobile device. Ergo, the behavior of this use case is currently unknown. We anticipate testing this capability in the next update and providing guidance.

C.1.3—Distribution

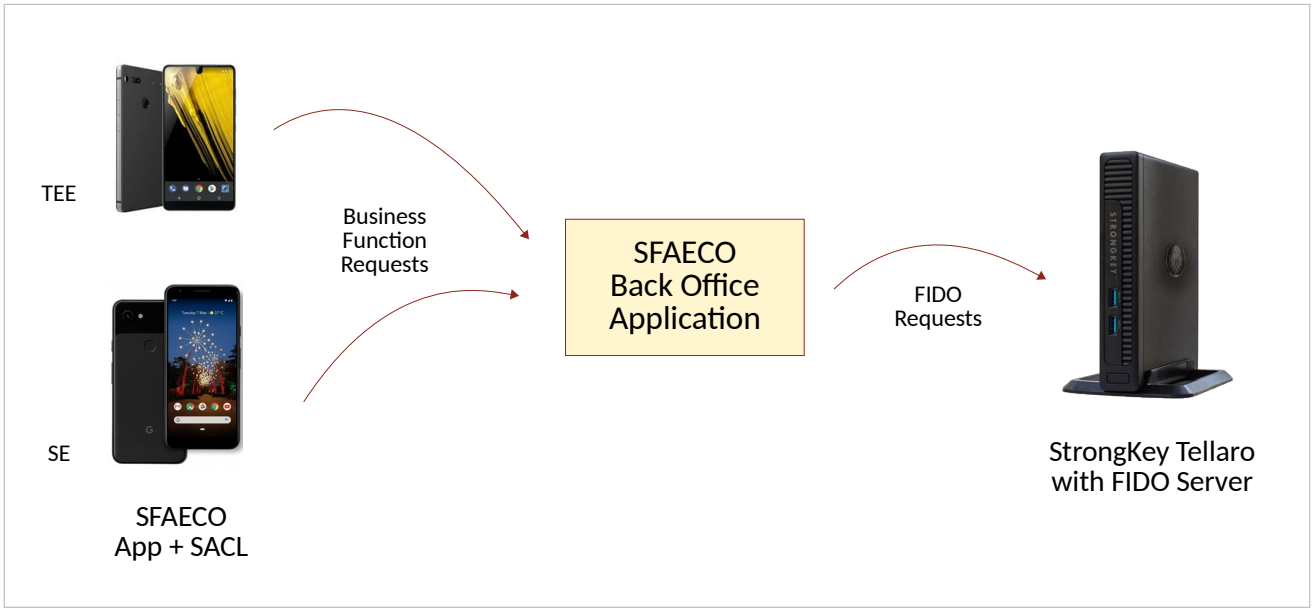
The distribution is available from Github and SourceForge.

When unzipped, the file expands to show the following folders and source code:

Folder Name	Notes
sfaeco	The NetBeans project that represents the server-side back end of the Sample FIDO App for E-commerce (SFAECO). The project is a Java Enterprise Edition 5 (JEE5) application written in Java8 using NetBeans 11.3 with OpenJDK 1.8.0_272.
skrepo	The folder with the Android Studio projects of the SACL, and the sample app (SFAECO) that uses the library. The projects were written in Java8, using OpenJDK 1.8.0_272 with Android Studio 3.6.1.
sfaeco/ sfaeco-client	The NetBeans module that represents command line (CLI) tools to test enrolling users. Not all web services of the application can be called by this CLI tool, since most of the web services require a FIDO Authenticator.
sfaeco/ sfaeco-ear	The NetBeans module that represents the Enterprise Archive. Its primarily a "container" project to build the `sfaeco-ear-1.0.ear` file which includes the necessary modules to run the backend application within a JEE5 Application Server (such as Payara).
sfaeco/ sfaeco-ejb	The NetBeans module that contains enterprise JavaBeans with the business logic and data model used by the sample application.
sfaeco/ sfaeco-web	The NetBeans module that contains servlet and exposes *Representational State Transfer (REST)* web services the Android app uses to interact with the application and the SKFS.
skrepo/ mobile/ android/ SampleSACL FidoEcoApp/	The Android sample app representing a simulated e-commerce application that demonstrates how to use SACL for the following functions: ▸ Enrolling Users ▸ Registering a FIDO2 key pair for the user ▸ Authenticating with the newly registered FIDO2 key ▸ Choosing one or more products from the Product Gallery to purchase ▸ Choosing a Payment Option to pay for the purchase transaction ▸ Confirming the transaction with a biometric response ▸ Seeing the result of a successful purchase transaction along with FIDO Alliance-EMVCo-defined data elements that can be relayed to issuing banks for authorization over 3DS (see the FIDO Alliance's FIDO Authentication and EMV 3-D Secure—Using FIDO for Payment Authentication white paper for details).
skrepo/ mobile/ android/ StrongKeyAndroidClient Library/	The Android client library representing a FIDO Authenticator that uses native Android APIs to support use of the FIDO2 protocol for registration, authentication, and transaction authorization with SKFS. The client library uses the following Android capabilities: ▸ *AndroidKeystore*, using either the Trusted Execution Environment (TEE) or a Secure Element (SE), depending on what is available on the phone ▸ *BiometricPrompt* to verify the user before enabling the use of *AndroidKeystore* for key generation and key usage with FIDO operations ▸ *RoomDB* to store Authenticator data locally on the device

C.1.4—Architecture of the Sample Application

The application and its back end have the following high-level architecture:



The app and SACL have been tested on the Essential PH-1 phone with Android 9; this device has a TEE within the device.

The app and SACL have also been tested on Google Pixel 3a and Google Pixel 4a phones, which have the Google Titan chip (SE) embedded within the device.

The app communicates over TLS to the SFAECO *Back Office Application (BOA)*, consuming REST web services running by default on StrongKey's PSD2 DEMO server on the internet at https://psd2demo.strongkey.com.

The StrongKey Tellaro also has the latest SKFS release running on the same appliance. While the DEMO uses the StrongKey Tellaro, SKFS can be deployed within virtual machines if desired (with some risk to the integrity of stored objects within the database of the FIDO server; please contact StrongKey if you wish to learn more about the risks of deploying key management solutions in multi-tenant public cloud environments).

C.1.5—Building the App

Once the distribution is downloaded and verified, unzip SampleFIDOAndroidApp.tgz in a location where you normally work with your Android projects.

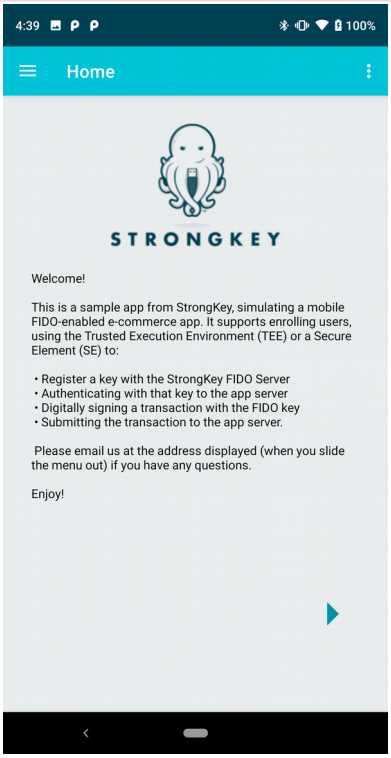
[sha256sum 08a4059c6326fbda30cc244ac5e532434a6ef8057f1aac81d413a87aa16bcc54]

- **Open** the *SampleSACLFidoEcoApp* project
- The `settings.gradle` file for the *SampleSACLFidoEcoApp* project must be updated: the ***projectDir*** variable must reflect where the SACL folder can be found locally on the PC, and then Gradle will connect
- Do not update the **biometric API**; it should be set to **1.0.1**
- Wait for Android Studio to **synchronize all files** it needs. When completed, **rebuild the app**

- **Connect** your Android 9+ phone to your development PC; this assumes your phone is already setup in Developer Mode
- **Launch** the app on your phone. Make sure Android Studio is showing Logcat messages and is filtered on the SFAECO app

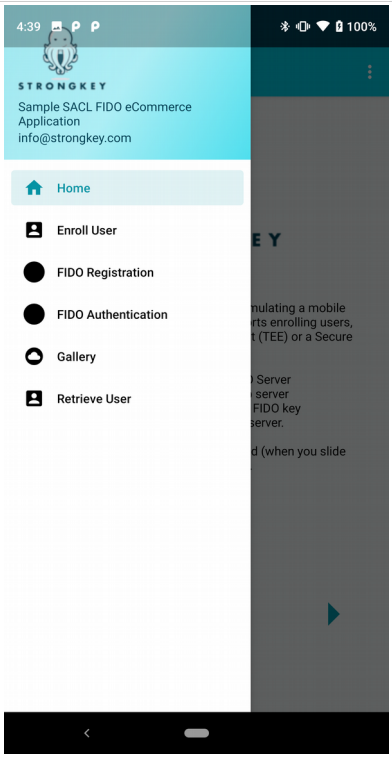
You should start seeing the following screens on your phone if all goes well:

1



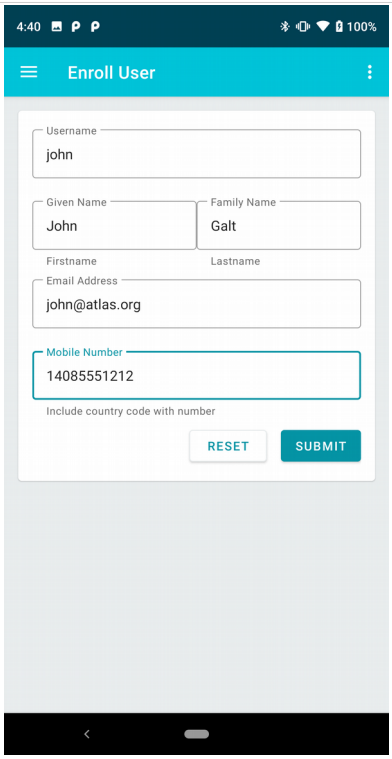
The **Home Page** of the app with a Welcome message.

2



The drawer Menu of the app that slides out from the left

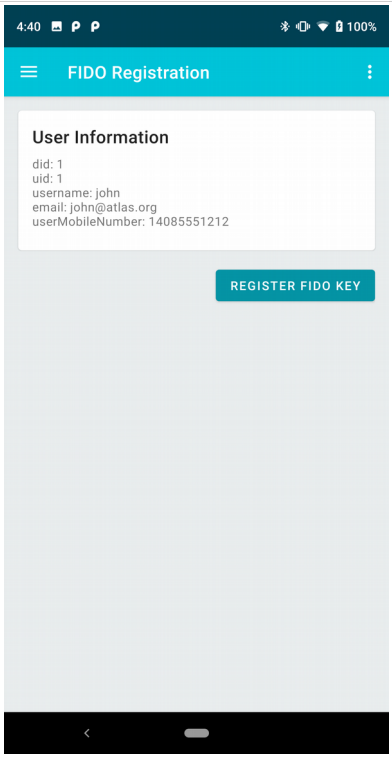
3



The **Enroll User** page where you can create a new user account.

None of the information provided need be real/authentic; the only validation performed is to check the uniqueness of the username, e-mail address, and mobile phone number on the server side—but they can all be fake information.

4



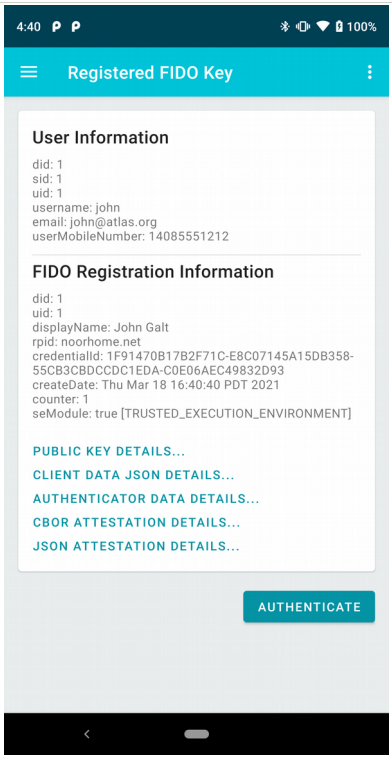
A successfully enrolled user.

Once enrolled, select the **REGISTER FIDO KEY** button to generate your key pair.

If for any reason, you fail to register a key (usually because the network goes to sleep and the TLS connection times out on the app), you can restart the app and choose **FIDO Registration** from the menu. The app will automatically recall the user information last saved on the mobile device.

If by any chance it does have a registered key, information about the registered FIDO key will also be retrieved from RoomDB and displayed.

5



A FIDO key successfully registered with the FIDO server.

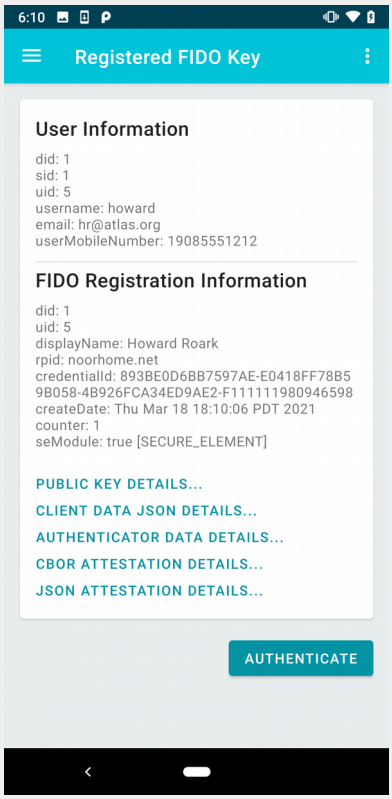
The page displays a fair amount of useful information about the FIDO key and security capability of the mobile device:

- The *relying party ID (RPID)* against which the key is registered
- The credential ID of the FIDO key
- The security capability of the mobile device—in this case, TEE

Each of the highlighted lines in blue provides more detailed information; most of it is not humanly readable, but it is available for developers to see if there is any interest.

Select the **AUTHENTICATE** button to authenticate with the newly registered key to the back-end application.

5A



This image shows the same information as the previous image—with one difference. Since this app was executing on a phone with a **SECURE ELEMENT (SE)**, the SACL detected this and displays this information when a key is registered.

> **NOTE:** In reality, the SACL does not make this determination. When the FIDO keys are generated, an attestation is also generated by *AndroidKeystore*; this attestation—the Android Key Attestation—generates a chain of X.509 digital certificates rooted in the hardware by the manufacturer of the device (in this case, Google, the manufacturer of the Pixel 3a).

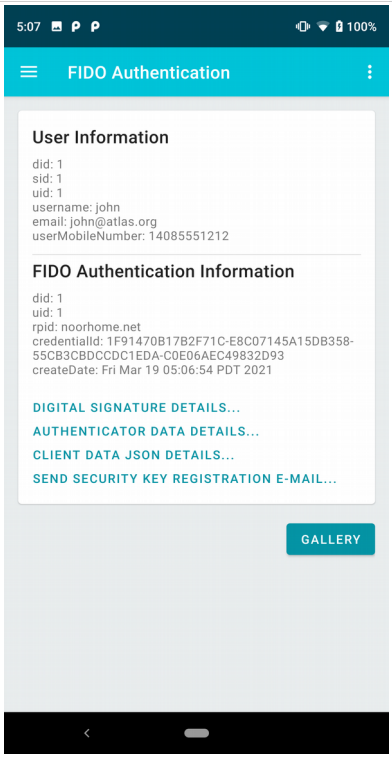
The chain of digital certificates provides information that is parsed by SKFS to determine properties of the public key that was sent for registration and the device in which the key was generated.

That information is relayed back to SACL, which is displayed by the app. This attestation is the assurance a *relying party (RP)* needs to affirm that they are dealing with an authentic device from the manufacturer and the attestation the manufacturer makes about cryptographic keys within that device.

While far more security capabilities are possible based on this technology, this SACL Preview Release is focused on enabling just what the FIDO2 protocol needs to satisfy business requirements.

StrongKey is committed to taking this technology and capability to its fullest potential, delivering some of the strongest security capability with its Tellaro appliance and securing some of the highest-risk business transactions. Please contact us if you would like to learn more.

6



A successfully authenticated user.

The page displays some useful information about the authentication.

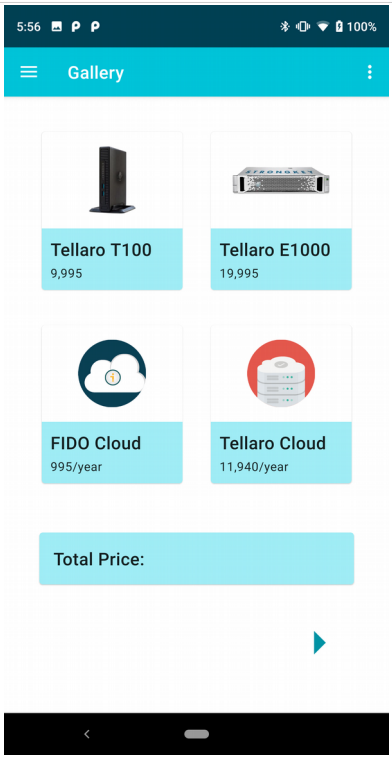
One useful capability (currently not implemented in this Preview) is a prompt to *SEND SECURITY KEY REGISTRATION E-MAIL*.

When a user registers with a site using their mobile device, this FIDO key becomes the strongest way in which the user may interact with the site. Once authenticated, it will be helpful to allow users to email themselves a one-time, time-limited link that allows them to register a second FIDO key using an external *Security Key*.

This has the advantage of allowing the user to use the site from their desktop or laptop, or to allow them to recover their account with a new phone if they lose their current mobile device.

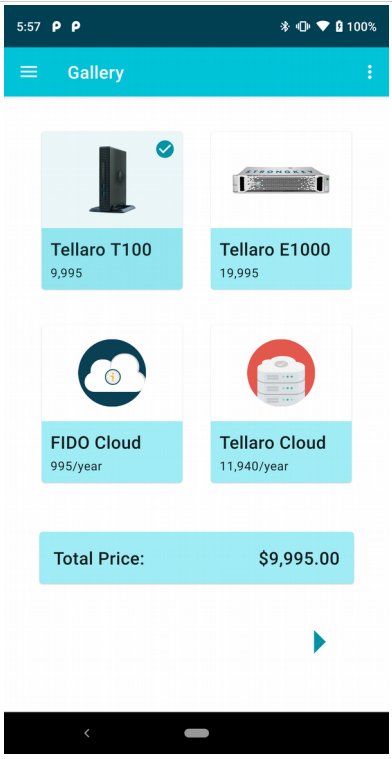
Once authenticated, select the **GALLERY** button to choose a sample product for purchase.

7



The **Product Gallery** displays 4 sample products in tiles that can be selected (or deselected, once selected) as part of the interaction.

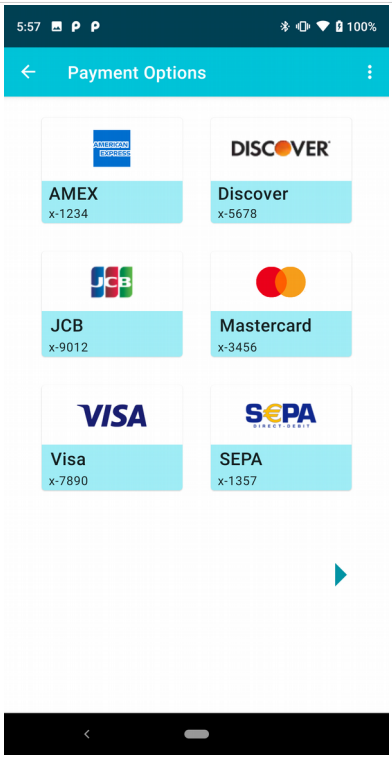
8



This page shows one of the tiles selected in this demonstration.

The arrow at the bottom right advances to next page.

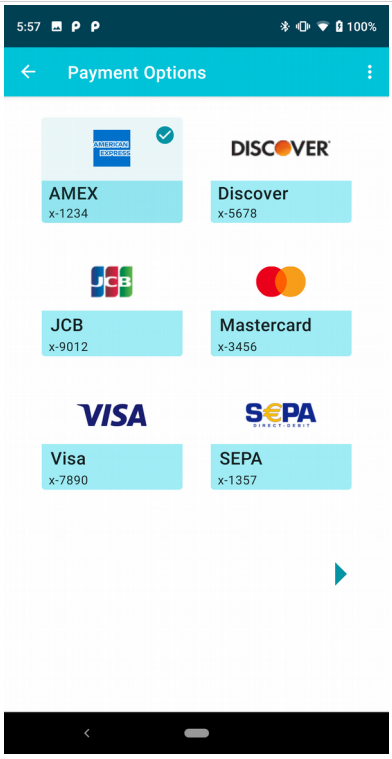
9



The **Payment Options page** displays six sample payment cards in tiles that can be selected\deselected as part of the interaction.

This assumes the app was designed with the capability to store multiple payment options with this site. In this sample app, hard-coded values are used to simulate this function.

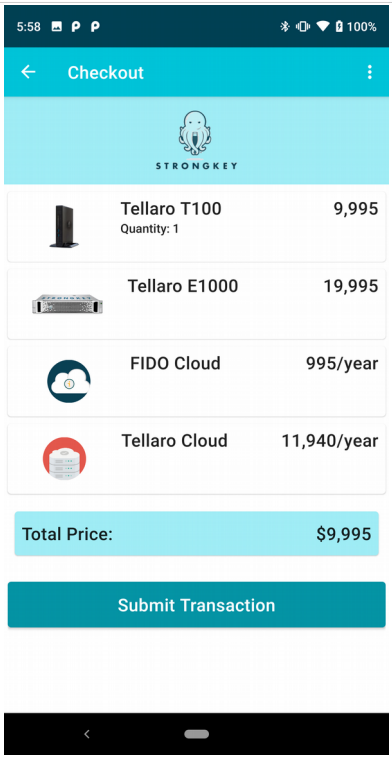
10



This page shows one of the tiles selected in this demonstration.

The arrow at the bottom right advances to next page.

11

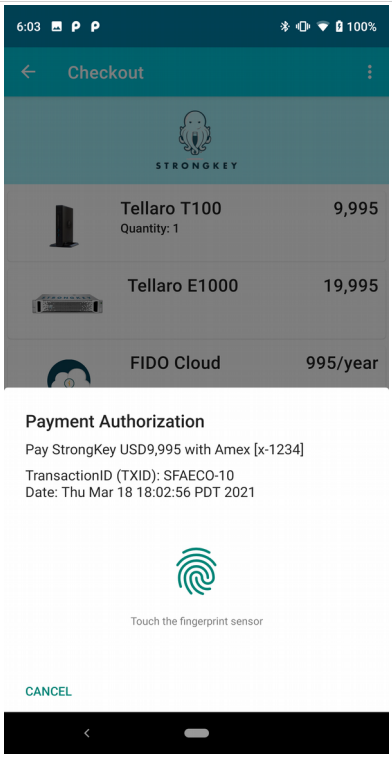


We finally arrive at the **CHECKOUT** page of the sample app.

It shows the which of the four products was chosen from the Product Gallery, and the *Total Price* the user is expected to pay to consummate this transaction.

The **SUBMIT TRANSACTION** is where the magic of FIDO *Transaction Authorization (TXA)* begins within SACL.

12



Having selected the **SUBMIT TRANSACTION** button, the app takes relevant information about the transaction and sends it to the back office application.

The back office application, in turn, processes that information and acquires a unique challenge from the FIDO Server, which takes this transaction's details into account.

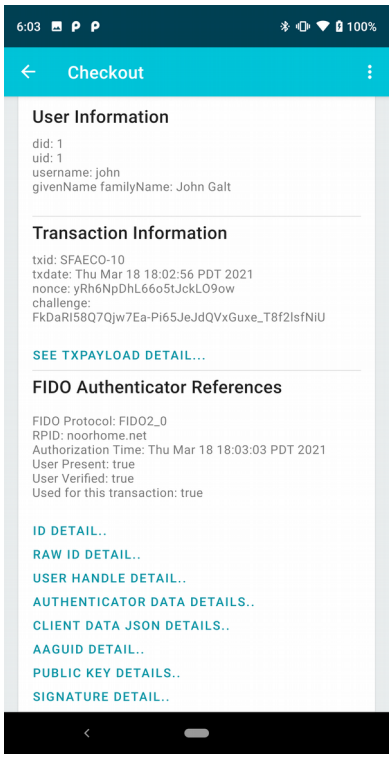
Upon receiving that challenge, the app calls Android's *BiometricPrompt* to display a unique message—a *dynamic link* in PSD2 RTS parlance—that displays necessary information for regulator compliance:

- Payee information
- The amount to be paid, and
- The chosen payment option

The *transaction ID (TXID)* and a Timestamp are added by the app to uniquely identify this transaction within its database.

The user is explicitly prompted to supply their fingerprint to authenticate with the device using the FIDO key to digitally sign the challenge sent by the FIDO server (through the back office application).

13



Upon successfully authenticating to the Android phone, the app—using SACL—uses the FIDO key to digitally sign the unique challenge and confirms the transaction displayed on the secure display.

This page shows the successful transaction with 2 interesting pieces of information:

- The SEE TXPAYLOAD DETAILS is Base64-encoded data of a JSON object with the following details:
  - Merchant name
  - Currency type
  - Total price
  - Card brand
  - Last 4 digits of the card
  - The unique transaction ID
  - The date/time of the transaction
- The information at the bottom, referenced as **FIDO Authenticator References**, refers to data elements standardized by the FIDO Alliance and EMVCo to transmit FIDO-authenticated transaction confirmations over 3DS messages to Issuing Banks for authorization. The information displayed here maps to the details specified in the FIDO Alliance's FIDO Authentication and EMV 3-D Secure—Using FIDO for Payment Authentication white paper.

This concludes the demonstration of the SFAECO app and its use of SACL for using FIDO2 protocols.

SACL can be used for applications in finance, healthcare, education, government, gaming, enterprise apps, etc. While mobile devices have made it significantly easier for users to authenticate to websites (by using biometrics), behind the curtain they still use the ancient password-based authentication schemes that are susceptible to attack and are responsible for more than 95% of all data breaches.

With SACL and SKFS, Android apps can now forever leave passwords behind. We hope you find this opportunity exciting to protect your users, as well as your own sites, by eliminating passwords and all the hassles that come with them.

Drop us a note at getsecure@strongkey.com to tell us what you think.

Enjoy!

C.2—JSON Web Tokens (JWTs)

SKFS now returns a *JSON web token (JWT)* upon authentication. A JWT is a self contained token that can be used between parties to provide trust. When a user requests an authentication with a *relying party (RP)*, the RP sends a request to the FIDO server. The FIDO server then verifies the user's identity based on the FIDO2 protocol. If the user identity is authenticated then the FIDO server creates a JWT and returns it to the RP as part of the response. Then the RP returns the JWT to the user. This user is then able to use the JWT for a predetermined length of time (default 30 minutes). Also, there is a independent program used for verifying the JWTs. This program is used by any service that wants to accept the FIDO server JWT as proof of identity. Upon receiving the request the service will pass the JWT, along with other data about the user, to the JWT verification program. That will then return whether or not the JWT is valid for the user. The advantage of using the JWT is it allows the user to only require verifying their identity with FIDO once per session while allowing them to access any service that has been configured to accept and verify the FIDO server's JWTs.

Encoded in the JWT is information about the user and their platform that serves to verify the user session during validation. By default the required content includes the relying party ID (`rpid`), time issued at (`iat`), expiration time (`exp`), client IP address (`cip`), username (`uname`), and client agent (`agent`). This information is digitally signed by node-specific signing keys. Each node in the cluster receives a set number of the signing keys to use for signing the JWTs it produces. This digital signature is then verified by the JWT verification program at the service the user is attempting to access.

Some required configurations are used in the creation of the JWTs.

The configuration file is located at `/usr/local/strongkey/crypto/etc/crypto-configuration.properties`. Here are the default configurations:

truststorelocation	Location of the truststore that holds the JWT signing certificates `crypto.cfg.property.jwtsigning.truststorelocation=/usr/local/strongkey/skfs/keystores/jwtsigningtruststore.bcfks`
password	Password used to access the keystore, TrustStore, and signing keys `crypto.cfg.property.jwtsigning.password=Abcd1234!`
threads	Number of threads that should be used for signing JWTs `crypto.cfg.property.jwtsigning.threads=10`
keystorelocation	Location of the keystore that has the JWT signing keys `crypto.cfg.property.jwtsigning.keystorelocation=/usr/local/strongkey/skfs/keystores/jwtsigningkeystore.bcfks`
algorithm	The cryptographic algorithm that should be used to sign the JWT `crypto.cfg.property.jwtsigning.algorithm=SHA256withECDSA`
certsperserver	Number of unique signing certificates to be allotted to each node `crypto.cfg.property.jwtsigning.certsperserver=3`

For the JWT to contain accurate information about the user two new parameters must be sent in the authentication request metadata: *clientIp* and *clientUserAgent*. *clientIp* must be an IP address used to access the relying party. *clientUserAgent* must be the user agent acessing the RP.

The JWT verification program requires the following the inputs: domainID, JWT, username, user agent, client ip, TrustStore password, TrustStore location, and the relying party ID. The domainID refers the ID of the cryptographic domain used to generate the JWT. User agent must be the user agent being used by the user to access the service. The client IP must be the the IP address of the user. The TrustStore location is where the truststore containing the JWT signing certificates is located. The the RPID must be the RPID that describes the RP through which the user received the JWT.

STRONGKEY

20045 Stevens Creek Boulevard Suite 2A
Cupertino, CA 95014
USA

i Apple iCloud: https://support.apple.com/en-us/HT202303
AWS S3: https://aws.amazon.com/s3

ii IBM Cloud HSM: http://www.softlayer.com/IBM-cloud-HSM
AWS Cloud HSM: https://aws.amazon.com/cloudhsm/

iii Infrastructure-as-a-Service: https://en.wikipedia.org/wiki/Cloud_computing#Infrastructure_as_a_service_.28IaaS.29

iv Washington Post: Hacked Dropbox login data of 68 million users is now for sale on the dark Web: https://www.washingtonpost.com/news/the-switch/wp/2016/09/07/hacked-dropbox-data-of-68-million-users-is-now-or-sale-on-the-dark-web/

v Payment Card Industry Data Security Standard:
https://www.pcisecuritystandards.org/document_library

vi Presentation on Suite B Cryptography, Elaine Barker, March 22, 2006:
http://csrc.nist.gov/groups/SMA/ispab/documents/minutes/2006-03/E_Barker-March2006-ISPAB.pdf

vii NIST Special Publication 800-132: Recommendation for Password-Based Key Derivation: http://dx.doi.org/10.6028/NIST.SP.800-132

viii About XEN Snapshots—Disk and memory snapshots: http://docs.citrix.com/en-us/xencenter/6-1/xs-xc-vms-snapshots/xs-xc-vms-snapshots-about.html
Live Migrating QEMU-KVM Virtual Machines:
http://developers.redhat.com/blog/2015/03/24/live-migrating-qemu-kvm-virtual-machines/
Export and Import Hyper-V virtual machines: https://msdn.microsoft.com/en-us/virtualization/hyperv_on_windows/user_guide/export_import
Moving or copying a virtual machine within a VMware environment (1000936):
https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1000936

ix Grayhat Warfare: https://buckets.grayhatwarfare.com/ and
How to search for Open Amazon s3 Buckets and their contents:
https://grayhatwarfare.medium.com/how-to-search-for-open-amazon-s3-buckets-and-their-contents-https-buckets-grayhatwarfare-com-577b7b437e01

x Flip Feng Shui: Hammering a Needle in the Software Stack, 25th USENIX Security Symposium, August 2016: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_razavi.pdf

xi NIST Cryptographic Module Validation Program:
https://csrc.nist.gov/Projects/Cryptographic-Module-Validation-Program

xii Public Key Cryptography Standard #11: http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm

xiii The New York Times: Capital One Data Breach Compromises Data of Over 100 Million: https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html

xiv FIDO Alliance: http://fidoalliance.org

xv FIDO Registration—U2F: https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-u2f-overview.html#registration-creating-a-key-pair
FIDO Registration—UAF: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-overview-v1.0-ps-20141208.html#authenticator-registration

xvi FIDO Authentication—U2F: https://fidoalliance.org/specs/fido-u2f-v1.0-nfc-bt-amendment-20150514/fido-u2f-overview.html#authentication-generating-a-signature
FIDO Authentication—UAF: https://fidoalliance.org/specs/fido-uaf-v1.0-ps-20141208/fido-uaf-overview-v1.0-ps-20141208.html#authentication

xvii A fully homomorphic encryption scheme, Craig Gentry, September 2009: https://crypto.stanford.edu/craig/craig-thesis.pdf

xviii Build a regulatory compliant web application, Arshad Noor, March 2012: http://www.ibm.com/developerworks/cloud/library/cl-regcloud/

Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 (General Data Protection Regulation): https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&rid=1

xix Regulation (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 (Payment Services Directive 2): https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32015L2366&rid=1
The White House Cybersecurity National Action Plan: https://www.whitehouse.gov/the-press-office/2016/02/09/fact-sheet-cybersecurity-national-action-plan
National Cyber Security Strategy 2016 to 2021: https://www.gov.uk/government/publications/national-cyber-security-strategy-2016-to-2021

xx FIDO Alliance—PSD2 Compliance: https://fidoalliance.org/psd2-compliance/

xxi InfoQ—Automating Data Protection Across the Enterprise: https://www.infoq.com/articles/cloud-data-encryption-infrastructure/