# FairPlay Streaming Server SDK

## Key Server Module Guide for Swift

v5.1

# Contents

## Introduction

Apple FairPlay Streaming (FPS) is a protocol to securely deliver content keys and their policies from a key server module (KSM) to client devices to enable playback of FairPlay Streaming protected content. To learn more about FPS and to download the latest software development kit (SDK), see https://developer.apple.com/streaming/fps/.

Apple provides registered FPS developers with an SDK that contains reference material, code, and tools to support FPS development. The SDK is available in Rust and Swift for macOS and Linux. You can choose to build your server using either operating system in the steps below. The Swift examples use an Apache HTTPD server and the Rust examples use a Rust HTTP server. Optionally, you may obtain additional tools and test streams from Apple that support the creation and testing of encrypted HLS streams.

This guide covers the KSM portion of the server SDK. For information about the client portion, refer to the `README.md` files in the `Client` folder.

The `Key Server Module` folder contains the following items:

- The KSM reference.
- A precompiled library with a corresponding header file. The library performs sensitive cryptographic operations and is only available in binary format. The library is compiled for macOS (universal binary) and Linux (x86_64 and arm64), and exposes the following function: `FPSStatus KSMCreateKeyPayload(KSMKeyPayload* ksmPayload);`
- A set of development credentials:
  - The FPS certificate, for both 1024- and 2048-bit certificates.
  - The FPS private keys, for both 1024- and 2048-bit certificates.
  - The FPS provisioning data.
- Server playback contexts (SPCs) created using development credentials.
- A content key context (CKC) verification tool for use with the development test vectors.

**Important:** You can use development credentials only to test sample key requests. They don't work with Apple devices.

## 1. Build your server for testing on Linux

This process builds the FairPlay Streaming key server module with test credentials, builds a sample http server, runs test cases, and verifies output.

Before starting, ensure the following:

1. You install Swift 6. See https://www.swift.org/install/ for instructions.

### Build the library

In Terminal, use the following commands to compile the Swift library:

```
cd Development/Key_Server_Module/Swift

swift build –Xbuild–tools–swiftc –DTEST_CREDENTIALS
```

After building the library, run test cases to ensure the library behaves as you expect. You can only use the provided test cases with the provided development credentials. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command runs all test inputs:

- If using an x86_64 machine:

```
export LD_LIBRARY_PATH=./Sources/prebuilt/x86_64–unknown–linux–\
gnu/
```

- If using an arm64 machine:

```
export LD_LIBRARY_PATH=./Sources/prebuilt/aarch64–unknown–linux–\
gnu/
```

Run all tests:

```
swift test –Xbuild–tools–swiftc –DTEST_CREDENTIALS \
––disable–swift–testing
```

Optionally, you can run a test on a single input, as in the following example:

```
swift run –Xbuild–tools–swiftc –DTEST_CREDENTIALS fpssdk_local \
../Test_Inputs/iOS/spc_ios_hd_lease_2048.json
```

### Build the server

We present two sample http server options. The first option uses Apache HTTPD with the FairPlay Streaming key server module built as a dynamic library that exposes a C-compatible API, and uses a C wrapper to make an Apache HTTPD server module. The second option uses Vapor with native Swift code.

## Build the server: Apache

**Note:** The sample commands below assume Red Hat Universal Base Image 9. For other Linux distributions, you may need to adjust the commands.

Before starting, ensure the following:

1. You install Apache HTTPD and the dev tools. Using the following command for installation:

   ```
   yum install httpd httpd-devel redhat-rpm-config
   ```

After you build the Swift library for testing, you can integrate it into the Apache server environment. Use the following command to build the server using `apxs`:

- If using an x86_64 machine:

```
   apxs -i -a -c \
-Wl,-L${PWD}/.build/x86_64-unknown-linux-gnu/debug/ \
-Wl,-lswift_fpssdk \
-Wl,-L${PWD}/Sources/prebuilt/x86_64-unknown-linux-gnu -lfpscrypto \
-Wl,-R${PWD}/.build/x86_64-unknown-linux-gnu/debug \
server_setup/mod_fps.c
```

- If using an arm64 machine:

```
   apxs -i -a -c \
-Wl,-L${PWD}/.build/arm64-unknown-linux-gnu/debug/ \
-Wl,-lswift_fpssdk \
-Wl,-L${PWD}/Sources/prebuilt/aarch64-unknown-linux-gnu \
-lfpscrypto \
-Wl,-R${PWD}/.build/arm64-unknown-linux-gnu/debug \
server_setup/mod_fps.c
```

Next, copy the dependent libraries to the Apache modules folder using these commands:

- If using an x86_64 machine:

```
   cp Sources/prebuilt/x86_64-unknown-linux-gnu/libfpscrypto.so \
/usr/lib64/httpd/modules/libfpscrypto.so

   cp .build/x86_64-unknown-linux-gnu/debug/libswift_fpssdk.so \
/usr/lib64/httpd/modules/libswift_fpssdk.so
```

- If using an arm64 machine:

```
   cp Sources/prebuilt/aarch64-unknown-linux-gnu/libfpscrypto.so \
/usr/lib64/httpd/modules/libfpscrypto.so

   cp .build/arm64-unknown-linux-gnu/debug/libswift_fpssdk.so \
/usr/lib64/httpd/modules/libswift_fpssdk.so
```

### Configure Apache HTTPD

You can configure Apache HTTPD by adding the module and handler to your Apache HTTPD configuration (`/etc/httpd/conf/httpd.conf`). Note that the `apxs` command may automatically add the `LoadModule` line in the previous step.

```
Listen 8080

LoadFile /usr/lib64/httpd/modules/libfpscrypto.so

LoadFile /usr/lib64/httpd/modules/libswift_fpssdk.so

LoadModule fps_module /usr/lib64/httpd/modules/mod_fps.so

<Location "/fps">

    SetHandler fps_handler

</Location>
```

Copy the credentials to the Apache modules folder.

```
cp -r Sources/src/extension/credentials /usr/lib64/httpd/modules/

export FPS_CREDENTIALS_PATH=/usr/lib64/httpd/modules/credentials
```

### Run your server

You can run the Apache HTTPD server with the configured module by using the following command:

```
httpd -D FOREGROUND
```

## Build the server: Vapor

After building the Swift library for testing, you can build a Swift Vapor server. Use the following command to build and run the server:

```
   swift run -Xbuild-tools-swiftc -DTEST_CREDENTIALS \
fpssdk_server_vapor
```

## Test your server

To ensure the server behaves as you expect, use curl to send an SPC to the server. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command sends a test SPC to the test server:

```
// In a new terminal window:

curl -d @../Test_Inputs/iOS/spc_ios_hd_lease_2048.json \
localhost:8080/fps
```

Use the `parse_fps` utility to verify the server's output and ensure that it returns the expected CKC. See Verify the key server module output for instructions.

After testing is complete, you can clean the build artifacts by using the following command:

```
swift package clean
```

## 2. Build your server for testing in macOS

This process builds the FairPlay Streaming key server module with test credentials, builds a sample http server, runs test cases, and verifies output.

Before starting, ensure the following:

1. You install Swift 6. See https://www.swift.org/install/ for instructions.

### Build the library

In Terminal, use the following command to build the library for testing:

```
cd Development/Key_Server_Module/Swift

swift build –Xbuild–tools–swiftc –DTEST_CREDENTIALS
```

To allow the cryptographic library to run, use the following command:

```
   xattr –d com.apple.quarantine \
Sources/prebuilt/macOS/libfpscrypto.dylib
```

Next, run test cases to ensure the library behaves as you expect. You can only use the provided test cases with the provided development credentials. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command runs all test inputs:

```
   swift test –Xbuild–tools–swiftc –DTEST_CREDENTIALS \
––disable–swift–testing
```

Optionally, you can run a test on a single input, as in the following example:

```
   swift run –Xbuild–tools–swiftc –DTEST_CREDENTIALS fpssdk_local \
../Test_Inputs/iOS/spc_ios_hd_lease_2048.json
```

### Build the server

We present two sample http server options. The first option uses Apache HTTPD with the FairPlay Streaming key server module built as a dynamic library that exposes a C-compatible API, and uses a C wrapper to make an Apache HTTPD server module. The second option uses Vapor with native Swift code.

### Build the server: Apache

Before starting, ensure the following:

1. You install Apache HTTPD using Homebrew. See https://formulae.brew.sh/formula/httpd for instructions on installation.

After you build the library for testing, you can integrate it into the Apache server environment. Depending on whether you are using Apple silicon or Intel, you need to follow different steps.

- If you're using Apple silicon, use `apxs` to build the Apache module with the following command:

```
   apxs -i -a -c \
-Wl,-L${PWD}/.build/arm64-apple-macosx/debug/ \
-Wl,-lswift_fpssdk \
-Wl,-L${PWD}/Sources/prebuilt/macos -Wl,-lfpscrypto \
-Wl,-R${PWD}/.build/arm64-apple-macosx/debug \
server_setup/mod_fps.c
```

Next, copy dependent libraries to the Apache `modules` folder (default location: `/opt/homebrew/lib/httpd/modules/`).

```
   cp .build/arm64-apple-macosx/debug/libswift_fpssdk.dylib \
/opt/homebrew/lib/httpd/modules/

   cp Sources/prebuilt/macos/libfpscrypto.dylib \
/opt/homebrew/lib/httpd/modules/
```

- If you're using Intel, use `apxs` to build the Apache module using the following command:

```
   apxs -i -a -c \
-Wl,-L${PWD}/.build/x86_64-apple-macosx/debug/ \
-Wl,-lswift_fpssdk \
-Wl,-L${PWD}/Sources/prebuilt/macOS -Wl,-lfpscrypto \
-Wl,-R${PWD}/.build/x86_64-apple-macosx/debug \
server_setup/mod_fps.c
```

Next, copy dependent libraries to the Apache modules folder, which may be located at `/opt/homebrew/lib/httpd/modules/`.

```
   cp Sources/prebuilt/macos/libfpscrypto.dylib \
/opt/homebrew/lib/httpd/modules/

   cp .build/x86_64-apple-macosx/debug/libswift_fpssdk.dylib \
/opt/homebrew/lib/httpd/modules/
```

**Configure Apache HTTPD**

Add the module and handler to your Apache HTTPD configuration, which may be located at `/opt/homebrew/etc/httpd/httpd.conf`. Note that the `apxs` command may automatically add the `LoadModule` line in the previous step.

```
Listen 8080

LoadFile /opt/homebrew/lib/httpd/modules/libfpscrypto.dylib

LoadFile /opt/homebrew/lib/httpd/modules/libswift_fpssdk.dylib

LoadModule fps_module /opt/homebrew/lib/httpd/modules/mod_fps.so

<Location "/fps">

    SetHandler fps_handler

</Location>
```

### Run your server

You can run the Apache HTTPD server with the configured module by using the following command:

```
/opt/homebrew/opt/httpd/bin/httpd -D FOREGROUND
```

## Build the server: Vapor

After building the Swift library for testing, you can build a Swift Vapor server. Use the following command to build and run the server:

```
swift run -Xbuild-tools-swiftc -DTEST_CREDENTIALS \
fpssdk_server_vapor
```

## Test your server

After configuring the server, send a test SPC to the test server to ensure the server behaves as you expect. There are sample inputs available for testing in a folder named `Test_Inputs`. The following command sends a test SPC to the test server:

```
// In a new terminal window:

curl -d @../Test_Inputs/iOS/spc_ios_hd_lease_2048.json \
localhost:8080/fps
```

Use the `parse_fps` utility to verify the server's output and ensure that it returns the expected CKC. See Verify the key server module output for instructions on verifying the server output.

```
swift package clean
```

## 3. Obtain production credentials

To use FPS in production, you generate private keys and certificate signing requests (CSRs), submit the CSRs to the Apple Developer website, and download the resulting credentials. This process involves generating both 1024-bit and 2048-bit RSA keys, creating CSRs, and then obtaining a certificate bundle and provisioning data through the Apple Developer website. You need to integrate these files into your SDK before using the server in production. This guide explains how to copy the private keys and provisioning data into the appropriate locations within the SDK for Swift.

### Generate a private key and a certificate signing request

Before you can submit a request to obtain production credentials, you need to obtain private keys and CSRs. Use the commands below to create new 1024-bit and 2048-bit private keys. If you already have an existing 1024-bit private key used with FairPlay Streaming, you may reuse it and only create the 2048-bit one.

The following Terminal input is an example of how to create a 1024-bit RSA private key and a certificate signing request:

```
   openssl req -out csr_1024.csr -new -newkey rsa:1024 \
-keyout priv_key_1024.pem \
-subj /CN=SubjectName/OU=OrganizationalUnit/O=Organization/C=US
```

The following Terminal input is an example of how to create a 2048-bit RSA private key and a certificate signing request:

```
   openssl req -out csr_2048.csr -new -newkey rsa:2048 \
-keyout priv_key_2048.pem \
-subj /CN=SubjectName/OU=OrganizationalUnit/O=Organization/C=US
```

Keep your private keys in a safe and secure location. You will need them to deploy your FairPlay Streaming key server.

### Submit the request

After you create your CSRs, you need to use them to create your certificate bundle and provisioning data through the Apple Developer website.

1. Go to the Apple Developer website at: https://developer.apple.com/.
2. Click Certificates, IDs, & Profiles.
3. Log in.
4. Click the Certificates tab.
5. Click the Add button (+).
6. Select FairPlay Streaming Certificate, and then click the Continue button.
7. Select SDK 5.x.
8. Under 2048-bit Certificate, click Choose File and select your 2048-bit certificate signing request file.

9. Under 1024-bit Certificate, either select to reuse your previous certificate, or click Choose File and select your 1024-bit certificate signing request file.
10. Click Continue.
11. Click the Download button to download the `fps_bundle.zip` file.

## Receive certificate bundle and provisioning data

The `fps_bundle.zip` file contains:

- `fps_certificate.bin`: this file should be hosted on your servers. Clients will need to fetch this certificate and use it when making license requests for your key server.
- `provisioning_data.bin`: this file is used by the FairPlay Streaming key server at runtime. See the integration instructions below.

## Integrate credentials

The SDK uses three credentials at runtime. The FairPlay Streaming key server reads the data inside the following files at runtime:

- `priv_key_1024.pem`:

  Copy this file to `Swift/Sources/src/extension/credentials/priv_key_1024.pem`.

- `priv_key_2048.pem`:

  Copy this file to `Swift/Sources/src/extension/credentials/priv_key_2048.pem`.

- `provisioning_data.bin`:

  Copy this file to `Swift/Sources/src/extension/credentials/provisioning_data.bin`.

**Note:** If you prefer to load your credentials another way, you can edit the `getPrivateKey()` and `getProvisioningData()` functions in `Swift/Sources/extension/structures/extension.swift`.

## 4. Build your server for production on Linux

Before starting, ensure the following:

1. You obtain production credentials and integrate the credentials into your library.
2. You install Swift 6. See https://www.swift.org/install/ for instructions.

### Build the library

Use the following command to build the library for production:

```
swift build -c release
```

### Build the server

We present two sample http server options. The first option uses Apache HTTPD with the FairPlay Streaming key server module built as a dynamic library that exposes a C-compatible API, and uses a C wrapper to make an Apache HTTPD server module. The second option uses Vapor with native Swift code.

### Build the server: Apache

**Note:** The sample commands below assume Red Hat Universal Base Image 9. For other Linux distributions, you may need to adjust the commands.

Before starting, ensure the following:

1. You install Apache HTTPD and the dev tools on your system. Use the following command for installation:

   ```
   yum install httpd httpd-devel redhat-rpm-config
   ```

After you build the library, integrate it into the Apache server. Use `apxs` to build the Apache module using the following command:

- If you're using an x86_64 machine, use `apxs` to build the Apache module with the following command:

  ```
     apxs -i -a -c \
  -Wl,-L${PWD}/.build/x86_64-unknown-linux-gnu/release/ \
  -Wl,-lswift_fpssdk \
  -Wl,-L${PWD}/Sources/prebuilt/x86_64-unknown-linux-gnu -lfpscrypto \
  -Wl,-R${PWD}/.build/x86_64-unknown-linux-gnu/release \
  server_setup/mod_fps.c
  ```

- If you're using an arm64 machine, use apps to build the Apache module with the following command:

  ```
     apxs -i -a -c \
  -Wl,-L${PWD}/.build/arm64-unknown-linux-gnu/release/ \
  -Wl,-lswift_fpssdk \
  -Wl,-L${PWD}/Sources/prebuilt/aarch64-unknown-linux-gnu \
  -lfpscrypto \
  -Wl,-R${PWD}/.build/arm64-unknown-linux-gnu/release \
  server_setup/mod_fps.c
  ```

Next, copy the dependent libraries to the Apache `modules` folder:

- If using an x86_64 machine, copy the libraries using these commands

```
   cp Sources/prebuilt/x86_64-unknown-linux-gnu/libfpscrypto.so \
/usr/lib64/httpd/modules/libfpscrypto.so

   cp .build/x86_64-unknown-linux-gnu/release/libswift_fpssdk.so \
/usr/lib64/httpd/modules/libswift_fpssdk.so
```

- If using an arm64 machine, copy the libraries using these commands

```
   cp Sources/prebuilt/aarch64-unknown-linux-gnu/libfpscrypto.so \
/usr/lib64/httpd/modules/libfpscrypto.so

   cp .build/arm64-unknown-linux-gnu/release/libswift_fpssdk.so \
/usr/lib64/httpd/modules/libswift_fpssdk.so
```

### Configure Apache HTTPD

You can configure Apache HTTPD by adding the module and handler to your Apache HTTPD configuration (`/etc/httpd/conf/httpd.conf`). Note that the `apxs` command may automatically add the `LoadModule` line in the previous step.

```
Listen 8080

LoadFile /usr/lib64/httpd/modules/libfpscrypto.so

LoadFile /usr/lib64/httpd/modules/libswift_fpssdk.so

LoadModule fps_module /usr/lib64/httpd/modules/mod_fps.so

<Location "/fps">

    SetHandler fps_handler

</Location>
```

### Run your server

After building the library and configuring your server, you can run the Apache server for production using the following command:

```
httpd -D FOREGROUND
```

## Build the server: Vapor

After building the Swift library you can build a Swift Vapor server. Use the following command to build the server:

```
swift run fpssdk_server_vapor
```

## 5. Build your server for production in macOS

Before starting, ensure the following:

1. You obtain production credentials and integrate the credentials into your library.
2. You install Swift 6. See https://www.swift.org/install/ for instructions on installation.

### Build the library

Use the following command to build the Swift library for production:

```
swift build -c release
```

### Build the server

We present two sample http server options. The first option uses Apache HTTPD with the FairPlay Streaming key server module built as a dynamic library that exposes a C-compatible API, and uses a C wrapper to make an Apache HTTPD server module. The second option uses Vapor with native Swift code.

### Build the server: Apache

Before starting, ensure the following:

1. You install Apache HTTPD using Homebrew. See https://formulae.brew.sh/formula/httpd for instructions on installation.

After you build the library for production, you can integrate it into the Apache server environment. Depending on whether you're using Apple silicon or Intel, you need to follow different steps.

- If you're using Apple silicon, use `apxs` to build the Apache module with the following command:

  ```
  apxs -i -a -c \
  -Wl,-L${PWD}/.build/arm64-apple-macosx/release/ \
  -Wl,-lswift_fpssdk \
  -Wl,-L${PWD}/Sources/prebuilt/macOS -Wl,-lfpscrypto \
  -Wl,-R${PWD}/.build/arm64-apple-macosx/release \
  server_setup/mod_fps.c
  ```

  Next, copy the dependent libraries to the Apache `modules` folder (default location: `/opt/homebrew/lib/httpd/modules/`).

  ```
  cp Sources/prebuilt/macos/libfpscrypto.dylib \
  /opt/homebrew/lib/httpd/modules/

  cp .build/arm64-apple-macosx/release/libswift_fpssdk.dylib \
  /opt/homebrew/lib/httpd/modules/
  ```

- If you're using Intel, use `apxs` to build the Apache module using the following command:

```
   apxs -i -a -c \
-Wl,-L${PWD}/.build/x86_64-apple-macosx/release/ \
-Wl,-lswift_fpssdk \
-Wl,-L${PWD}/Sources/prebuilt/macOS -Wl,-lfpscrypto \
-Wl,-R${PWD}/.build/x86_64-apple-macosx/release \
server_setup/mod_fps.c
```

  Next, copy the dependent libraries to the Apache `modules` folder (default location: `/opt/homebrew/lib/httpd/modules/`).

```
   cp Sources/prebuilt/macos/libfpscrypto.dylib \
/opt/homebrew/lib/httpd/modules/

   cp .build/x86_64-apple-macosx/release/libswift_fpssdk.dylib \
/opt/homebrew/lib/httpd/modules/
```

#### Configure Apache HTTPD

Add the module and handler to your Apache HTTPD configuration (default location: `/opt/homebrew/etc/httpd/httpd.conf`). Note that the `apxs` command may automatically add the `LoadModule` line in the previous step.

```
Listen 8080

LoadFile /opt/homebrew/lib/httpd/modules/libfpscrypto.dylib

LoadFile /opt/homebrew/lib/httpd/modules/libswift_fpssdk.dylib

LoadModule fps_module /opt/homebrew/lib/httpd/modules/mod_fps.so

<Location "/fps">

    SetHandler fps_handler

</Location>
```

#### Run your server

After building the library and configuring your server, you can run the Apache server for production using the following command:

```
/opt/homebrew/opt/httpd/bin/httpd -D FOREGROUND
```

### Build the server: Vapor

After building the Swift library you can build a Swift Vapor server. Use the following command to build the server:

```
swift run fpssdk_server_vapor
```

## 6. Verify the key server module output

Before starting, ensure you install Python 3. See https://www.python.org/downloads/ for instructions. The FPS Server SDK package includes pre-generated SPC test vectors and a verification utility called `parse_fps` to test your KSM implementation. This utility takes in the Base64 or binary-encoded SPC and, optionally, the corresponding CKC field (Base64 or binary-encoded format), and then decrypts and prints the data within them.

1. Install the crypto package for python:

   ```
   pip3 install –U PyCryptodome
   ```

2. Copy one `libfpscrypto` for whichever architecture `parse_fps` is running on into the `parse_fps/` directory.

   ```
   cp Swift/Sources/prebuilt/macos/libfpscrypto.dylib parse_fps/

   cp Swift/Sources/prebuilt/x86_64–unknown–linux–gnu/\
   libfpscrypto.so parse_fps/
   ```

3. Run the following command:

   ```
   cd Development/Key_Server_Module/parse_fps

   python3 –m fps.parse_fps ––spc ./samples/sample_spc.b64 \
   ––ckc ./samples/sample_ckc.b64
   ```

You can find a sample output using the SPC and CKC in `parse_fps/samples:`.

### Add production credentials

You can use production credentials to verify input and output using `parse_fps` as well. To add production credentials, add extra fields to the `fps/cfg/credentials_sdk.py` file in the following format:

```
{

    'pkey_1024': 'credentials/priv_key_1024.pem',

    'pkey_2048': 'credentials/priv_key_2048.pem',

    'cert': 'credentials/fps_certificate.bin',

    'provisioning_data': 'credentials/provisioning_data.bin',

}
```

After adding the credentials, you can use `parse_fps` to test and verify production input and output. The format for using the tool is the same as when using test credentials. You don’t need to remove the fields for test credentials when adding production credentials.

## 7. Customize the key server module

The key server module has an extension component in the SDK that you can modify for additional functionality and customization. The extension is where you add any custom code, structures, variables, and logic.

Many of the Base functions call a corresponding `Custom` function to allow for customization in the extension. These extension functions are located in `Swift/Sources/src/extension/structures/extension.swift`.The file contains functions that allow you to change the behavior at many points of the server code, such as handling additional fields in the input JSON file, or adding additional fields to the output JSON file.

### Logging

Calls to the key server module result in the creation of an output JSON file. This output file may contain a successful status code and the resulting license, or it may contain an error code. To help with debugging error scenarios, by default, the KSM also prints log messages. Throughout the code, there are two main types of logging: production and debug. Both allow you to customize the format.

Production logs:

- Function name: `fpsLogError()`
- Enabled in debug and release builds.
- Print to `stderr`.
- Print with a prefix (timestamp, name, version, file, function, line number), which you can customize in `src/logging.swift`.

Debug logs:

- Function name: `Log.Debug()`
- Disabled in release builds.
- Print to stderr.
- You can customize the format in `src/logging.swift`.

### Determine your business rules

The `checkBusinessRules()` function provides a suggested set of business rules that includes the following:

- UHD content requires security level Main and HDCP Type 1.
- HD content requires security level Baseline or higher, and HDCP Type 0.
- SD content requires security level Baseline or higher.
- Audio content has no special requirements.

You can edit or add rules to match your business requirements. For example, if your rules need additional input types, or you have different requirements for 720 p and 1080 p, or SDR and HDR, you can add types to the `content-type` field in the input JSON file, which the `ContentType` enumeration describes and the `parseAssetInfoCustom()` function parses.

## Customize using extension structures

You can find custom structures in the `extension_structures` file. These structures are all originally empty, so you add fields for custom implementations. The structures are:

| Extension | Description |
| --- | --- |
| `SKDExtension` | A custom extension to modify how to deliver, validate, and store keys on the client side. |
| `FPSOperationExtension` | A custom extension to modify the set of operations that the SDK performs, such as playback or key requests. |
| `AssetInfoExtension` | A custom extension to modify information about the asset that's streaming, such as its encryption status, content ID, or custom attributes that the streaming app uses. |
| `ServerCtxExtension` | A custom extension to modify the server-side operations, such as license requests or validation of the server's identity. |
| `FPSResultsExtension` | A custom extension to modify the result of an operation, such as a license or playback request. |
| `FPSResultExtension` | A custom extension to modify individual operation results. |
| `SPCDataExtension` | A custom extension to modify the secure playback context data by customizing its validation for specific content playback scenarios. |
| `CKCDataExtension` | A custom extension to modify the content key context data by customizing how the key server processes and delivers key responses to the client. |
| `SPCContainerExtension` | A custom extension to modify the SPC container by defining additional encapsulation logic for secure key request data before sending it to the key server. |
| `CKCContainerExtension` | A custom extension to modify the CKC container by customizing content key response data for enhanced security. |
| `KeyDurationExtension` | A custom extension to modify key duration settings by defining specific playback duration rules. |

## 8. API reference

The key server module consists of two main components: `Base` and `Extension`. The `Base` structure contains all parts of the FairPlay Streaming logic that are necessary for the protocol to function and isn't intended to be modified. `Extension` allows for added functionality and customization and is intended to be modified. The Swift folder contains a `Sources` folder with Swift source files and a manifest file (`Package.swift`) that defines the package and its contents, and contains details about the build targets and their dependencies. The `Base` class contains the basic functionality to parse SPCs and create CKCs. Each part of this process is divided into a separate directory for better readability.

### Base structures

The following code is part of the `Base` structures in `base_fps_structures.swift`:

`FPSOperations`

| Swift | |
|---|---|
| `public struct FPSOperations: Codable {`<br>`  var operationsPtr: [FPSOperation]`<br>`}` | |
| API | Description |
| `public struct FPSOperations: Codable { }` | Contains a vector of `FPSOperation`. This is necessary if receiving multiple requests in the same JSON file. |
| `var operationsPtr: [FPSOperation]` | A list of `FPSOperation` objects. |

`FPSOperation`

| Swift | |
|---|---|
| `public struct FPSOperation: Codable {`<br>`  var id: UInt64`<br>`  var spc: [UInt8]`<br>`  var isCheckIn: Bool`<br>`  var assetInfo: AssetInfo`<br><br>`  public var ext: FPSOperationExtension`<br>`}` | |
| API | Description |
| `public struct FPSOperation: Codable { }` | The basic structure of a FairPlay Streaming key request after parsing the JSON file. Includes the ID of the request, the SPC, and the `AssetInfo`. |

| | |
|---|---|
| `var id: UInt64` | A unique ID for the operation request. |
| `var spc: [UInt8]` | An `Array` of the server playback context data. |
| `var isCheckIn: Bool` | A Boolean value that indicates whether this is a check-in request. |
| `var assetInfo: AssetInfo` | Information about the asset. |
| `public var ext: FPSOperationExtension` | A structure for extending `FPSOperation` with custom fields. |

### `AssetInfo`

Swift

```
public struct AssetInfo: Codable {
  var key: [UInt8]
  var iv: [UInt8]
  var isCKProvided: Bool
  var leaseDuration: UInt32
  var rentalDuration: UInt32
  var playbackDuration: UInt32

  public var hdcpReq:
      base_constants.FPSHDCPRequirement

  var licenseType: base_constants.FPSLicenseType
  var streamId: [UInt8]
  var titleId: [UInt8]  .
  var isStreamIdSet: Bool
  var isTitleIdSet: Bool

  public var ext: AssetInfoExtension
 }
```

| API | Description |
|---|---|
| `public struct AssetInfo: Codable { }` | Contains information about the requested asset. |
| `var key: [UInt8]` | The encryption key. |
| `var iv: [UInt8]` | The initialization vector. |
| `var isCKProvided: Bool` | A Boolean value that indicates whether the content key is provided. |
| `var leaseDuration: UInt32` | The license duration, in seconds, starting from SPC creation time. Required for lease requests. Mutually exclusive with `offline-hls`. |

| | |
|---|---|
| `var rentalDuration: UInt32` | The rental duration, in seconds, starting from asset download time. |
| `var playbackDuration: UInt32` | The lease duration, in seconds, starting from asset first playback time. |
| `public var hdcpReq: base_constants.FPSHDCPRequirement` | Specifies the High-bandwidth Digital Content Protection(HDCP) for the asset. |
| `var licenseType: base_constants.FPSLicenseType` | The license associated with the asset. |
| `var streamId: [UInt8]` | The stream ID associated with the asset. |
| `var titleId: [UInt8]` | The title ID associated with the asset. |
| `var isStreamIdSet: Bool` | A Boolean value that indicates whether the stream ID is set. |
| `var isTitleIdSet: Bool` | A Boolean value that indicates whether the title ID is set. |
| `public var ext: AssetInfoExtension` | A structure for extending `AssetInfo` with custom fields. |

### `FPSResults`

| Swift | |
|---|---|
| `public struct FPSResults: Codable { public var resultPtr: [FPSResult]`<br><br>`public var ext: FPSResultsExtension }` | |
| API | Description |
| `public struct FPSResults: Codable { }` | Contains a vector of `FPSResult`. This is necessary when sending multiple requests at once (much like `FPSOperations`). |
| `public var resultPtr: [FPSResult]` | A list of `FPSResult` objects. |
| `public var ext: FPSResultsExtension` | A structure for extending `FPSResults` with custom fields. |

## FPSResult

Swift

```
public struct FPSResult: Codable {
  public var id: UInt64
  public var status: FPSStatus
  public var hu: [UInt8]
  public var ckc: [UInt8]

  public var sessionId: UInt64

  public var isCheckIn: Bool
  public var syncServerChallenge: UInt64
  public var syncFlags: UInt64
  public var syncTitleId: [UInt8]
  public var durationToRentalExpiry: UInt32
  public var recordsDeleted: Int
  public var deletedContentIDs: [UInt8]

  public var deviceIdentitySet: Bool
  public var fpdiVersion: UInt32
  public var deviceClass: UInt32
  public var vendorHash: [UInt8]
  public var productHash: [UInt8]
  public var fpVersionREE: UInt32
  public var fpVersionTEE: UInt32
  public var osVersion: UInt32

  public var vmDeviceInfo: Optional<VMDeviceInfo>

  public var ext: FPSResultExtension
 }
```

| API | Description |
| --- | --- |
| `public struct FPSResult: Codable { }` | The structure containing the response information before the key server serializes it into a JSON file. |
| `public var id: UInt64` | A unique ID. |
| `public var status: FPSStatus` | The status of the operation. |
| `public var hu: [UInt8]` | An `Array` representing the HU identifier. |
| `public var ckc: [UInt8]` | An `Array` representing the content key context. |
| `public var sessionId: UInt64` | The session identifier. |
| `public var isCheckIn: Bool` | A Boolean value that indicates whether this is a check-in response. |
| `public var syncServerChallenge: UInt64` | A server challenge for synchronization. |
| `public var syncFlags: UInt64` | A flag that holds synchronization status. |

| | |
|---|---|
| `public var syncTitleId: [UInt8]` | An `Array` representing the synchronization title ID. |
| `public var durationToRentalExpiry: UInt32` | The duration remaining until the rental content expires, in seconds. |
| `public var recordsDeleted: Int` | The number of deleted records. |
| `public var deletedContentIDs: [UInt8]` | A list of deleted content IDs. |
| `public var deviceIdentitySet: Bool` | A Boolean value that indicates whether the device identity is successfully set. |
| `public var fpdiVersion: UInt32` | The current version of the FPDI. |
| `public var deviceClass: UInt32` | The type of device for streaming. |
| `public var vendorHash: [UInt8]` | A hash value representing the device vendor. |
| `public var productHash: [UInt8]` | A hash value representing the product. |
| `public var fpVersionREE: UInt32` | The FairPlay version in the Rich Execution Environment (REE). |
| `public var fpVersionTEE: UInt32` | The FairPlay version in the Trusted Execution Environment (TEE). |
| `public var osVersion: UInt32` | The operating system version running on the device. |
| `public var vmDeviceInfo: Optional<VMDeviceInfo>` | Host and guest device information, if running in a virtual machine. |
| `public var ext: FPSResultExtension` | A structure for extending `FPSResult` with custom fields. |

## KSMKeyPayload

Swift

```
typedef struct KSMKeyPayload {
   uint64_t version;
   const uint8_t* contentKey;
   uint64_t contentKeyLength;
   const uint8_t* contentIV;
   uint64_t contentIVLength;
   uint64_t contentType;
   const uint8_t* SK_R1;
   uint64_t SK_R1Length;
   const uint8_t* R2;
   uint64_t R2Length;
   const uint8_t* R1Integrity;
   uint64_t R1IntegrityLength;
   const uint64_t* supportedKeyFormats;
   uint64_t numberOfSupportedKeyFormats;
   uint64_t cryptoVersionUsed;
   const uint8_t* provisioningData;
   uint64_t provisioningDataLength;
   const uint8_t* certHash;
   uint64_t certHashLength;
   const uint8_t* clientHU;
   uint64_t clientHULength;
   uint64_t contentKeyTLLVTag;
   const uint8_t* contentKeyTLLVPayload;
   uint64_t contentKeyTLLVPayloadLength;
   const uint8_t* R1;
   uint64_t R1Length;
} KSMKeyPayload;
```

| API | Description |
| --- | --- |
| `typedef struct KSMKeyPayload { }` | A structure containing all the information to send to the cryptographic library to create the content key TLLV. |
| `uint64_t version;` | The version number. |
| `const uint8_t* contentKey;` | A pointer to the content key. |
| `uint64_t contentKeyLength;` | The length of the content key. |
| `const uint8_t* contentIV;` | A pointer to the content IV. |
| `uint64_t contentIVLength;` | The length of the content IV. |
| `uint64_t contentType;` | The type of content. |
| `const uint8_t* SK_R1;` | A pointer to session key R1. |
| `uint64_t SK_R1Length;` | The length of session key R1. |
| `const uint8_t* R2;` | A pointer to R2. |
| `uint64_t R2Length;` | The length of R2. |
| `const uint8_t* R1Integrity;` | A pointer to the R1 integrity value. |

| | |
|---|---|
| `uint64_t R1IntegrityLength;` | The length of the R1 integrity value. |
| `const uint64_t* supportedKeyFormats;` | A pointer to the list of supported key formats. |
| `uint64_t numberOfSupportedKeyFormats;` | The number of supported key formats. |
| `uint64_t cryptoVersionUsed;` | The version of the cryptographic library. |
| `const uint8_t* provisioningData;` | A pointer to the provisioning data. |
| `uint64_t provisioningDataLength;` | The length of the provisioning data. |
| `const uint8_t* certHash;` | A pointer to the certificate hash. |
| `uint64_t certHashLength;` | The length of the certificate hash. |
| `const uint8_t* clientHU;` | A pointer to the client HU. |
| `uint64_t clientHULength;` | The length of the client HU. |
| `uint64_t contentKeyTLLVTag;` | The tag associated with the content key TLLV. |
| `const uint8_t* contentKeyTLLVPayload;` | A pointer to the payload of the content key TLLV. |
| `uint64_t contentKeyTLLVPayloadLength;` | The length of the content key TLLV payload. |
| `const uint8_t* R1;` | A pointer to R1. |
| `uint64_t R1Length;` | The length of R1. |

The following code is part of the `Base` structures in `base_server_structures.swift`:

### `FPSServerCtx`

Swift

```swift
public struct FPSServerCtx {
  public var spcContainer: FPSServerSPCContainer
  public var ckcContainer: FPSServerCKCContainer
  public var isStreamIdSet: Bool
  public var streamId: [UInt8]
  public var isTitleIdSet: Bool
  public var titleId: [UInt8]

  public var ext: ServerCtxExtension
}
```

| API | Description |
|---|---|

| | |
|---|---|
| `public struct FPSServerCtx { }` | Contains the SPC and CKC containers, along with stream and title IDs. This is the base structure that the key server most commonly sends to functions. |
| `public var spcContainer: FPSServerSPCContainer` | A container that holds the SPC data. |
| `public var ckcContainer: FPSServerCKCContainer` | A container that holds the CKC data. |
| `public var isStreamIdSet: Bool` | A Boolean value that indicates whether the stream ID is set. |
| `public var streamId: [UInt8]` | The stream ID. |
| `public var isTitleIdSet: Bool` | A Boolean value that indicates whether the title ID is set. |
| `public var titleId: [UInt8]` | The title ID. |
| `public var ext: ServerCtxExtension` | A structure for extending `FPSServerCtx` with custom fields. |

### `FPSServerSPCContainer`

| Swift |
|---|

```swift
public struct FPSServerSPCContainer {
  public var version: UInt32
  public var reservedValue: UInt32
  public var aesKeyIV: [UInt8]
  public var aesWrappedKey: [UInt8]
  public var aesWrappedKeySize: Int
  public var certificateHash: [UInt8]
  public var spcDecryptedData: [UInt8]
  public var spcDataSize: Int
  public var spcDataOffset: Int
  public var spcData: FPSServerSPCData

  public var ext: SPCContainerExtension
}
```

| API | Description |
|---|---|
| `public struct FPSServerSPCContainer { }` | Contains all the parsed information from the SPC, including version, SPC encryption, AES key and IV, and TLLV data. |
| `public var version: UInt32` | Stores the current SPC version. |
| `public var reservedValue: UInt32` | Stores a reserved value. |

| | |
|---|---|
| `public var aesKeyIV: [UInt8]` | The AES key IV value. |
| `public var aesWrappedKey: [UInt8]` | The AES wrapped key value. |
| `public var aesWrappedKeySize: Int` | Stores the size of the AES wrapped key. |
| `public var certificateHash: [UInt8]` | The hash of the certificate value. |
| `public var spcDecryptedData: [UInt8]` | Stores the decrypted SPC data. |
| `public var spcDataSize: Int` | Stores the size of the SPC data. |
| `public var spcDataOffset: Int` | The SPC offset data. |
| `public var spcData: FPSServerSPCData` | The parsed SPC data. |
| `public var ext: SPCContainerExtension` | The additional extension data for the SPC container. |

## FPSServerCKCContainer

| Swift | |
|---|---|
| `public struct FPSServerCKCContainer {`<br>`  public var version: UInt32`<br>`  public var aesKeyIV: [UInt8]`<br>`  public var ckc: [UInt8]`<br>`  public var ckcDataPtr: [UInt8]`<br>`  public var ckcData: FPSServerCKCData`<br>`}` | |
| API | Description |
| `public struct FPSServerCKCContainer { }` | Contains information to add to the CKC, including version, CKC encryption, AES key and IV, and the CKC payload. |
| `public var version: UInt32` | Stores the version of the CKC. |
| `public var aesKeyIV: [UInt8]` | The key IV value. |
| `public var ckc: [UInt8]` | The CKC data as an `Array`. |
| `public var ckcDataPtr: [UInt8]` | Points to the CKC data as an `Array`. |
| `public var ckcData: FPSServerCKCData` | Contains the CKC data. |

`FPSServerSPCData`

Swift

```
public struct FPSServerSPCData {
  public var antiReplay: [UInt8]
  public var sk: [UInt8]
  public var hu: [UInt8]
  public var r2: [UInt8]
  public var r1: [UInt8]
  public var skR1IntegrityTag: [UInt8]
  public var skR1Integrity: [UInt8]
  public var skR1: [UInt8]
  public var assetId: [UInt8]
  public var versionUsed: UInt32
  public var versionsSupported: [UInt32]
  public var returnTLLVs: [FPSServerTLLV]
  public var returnRequest: FPSServerTLLV
  public var clientFeatures: FPSServerClientFeatures
  public var spcDataParser: FPSServerSPCDataParser
  public var playInfo: FPSServerMediaPlaybackState
  public var streamingIndicator: UInt64
  public var transactionId: UInt64

  public var syncServerChallenge: UInt64
  public var syncFlags: UInt64
  public var syncTitleId: [UInt8]
  public var durationToRentalExpiry: UInt32
  public var recordsDeleted: Int
  public var deletedContentIDs: [UInt8]

  public var clientCapabilities: [UInt8]

  public var isSecurityLevelTLLVValid: Bool
  public var supportedSecurityLevel: UInt64
  public var clientKextDenyListVersion: UInt32

  public var deviceIdentity: FPSDeviceIndentity
  public var deviceInfo: FPSDeviceInfo

  public var numberOfSupportedKeyFormats: UInt32
  public var supportedKeyFormats: [UInt64]

  public var vmDeviceInfo: Optional<VMDeviceInfo>

  public var ext: SPCDataExtension
}
```

| API | Description |
| --- | --- |
| `public struct FPSServerSPCData { }` | Contains parsed information from the SPC TLLVs after decryption. |
| `public var antiReplay: [UInt8]` | The antireplay value. |
| `public var sk: [UInt8]` | An `Array` that represents the session key for decryption. |
| `public var hu: [UInt8]` | An `Array` that represents the HU. |

| | |
|---|---|
| `public var r2: [UInt8]` | An `Array` that represents the r2. |
| `public var r1: [UInt8]` | An `Array` that represents the r1. |
| `public var skR1IntegrityTag: [UInt8]` | An `Array` that represents the integrity tag for session key R1. |
| `public var skR1Integrity: [UInt8]` | An `Array` that represents the integrity value for session key R1. |
| `public var skR1: [UInt8]` | An `Array` that represents the session key r1. |
| `public var assetId: [UInt8]` | The asset ID that identifies the asset. |
| `public var versionUsed: UInt32` | The version number. |
| `public var versionsSupported: [UInt32]` | A list of supported versions. |
| `public var returnTLLVs: [FPSServerTLLV]` | A list of TLLVs that the server returns. |
| `public var returnRequest: FPSServerTLLV` | A specific TLLV that the server returns as part of the request. |
| `public var clientFeatures: FPSServerClientFeatures` | The client features. |
| `public var spcDataParser: FPSServerSPCDataParser` | An instance for parsing the SPC data. |
| `public var playInfo: FPSServerMediaPlaybackState` | The current state of media playback. |
| `public var streamingIndicator: UInt64` | A streaming indicator for content. |
| `public var transactionId: UInt64` | A unique transaction identifier. |
| `public var syncServerChallenge: UInt64` | A challenge number for synchronizing the server. |
| `public var syncFlags: UInt64` | Flags for controlling synchronization settings. |
| `public var syncTitleId: [UInt8]` | An `Array` representing the synchronization title ID. |
| `public var durationToRentalExpiry: UInt32` | The duration remaining until rental expiration. |
| `public var recordsDeleted: Int` | The number of deleted records. |
| `public var deletedContentIDs: [UInt8]` | A list of deleted content IDs. |
| `public var clientCapabilities: [UInt8]` | An `Array` representing the client's capabilities. |

| | |
|---|---|
| `public var isSecurityLevelTLLVValid: Bool` | A Boolean value that indicates whether the security level TLLV is valid. |
| `public var supportedSecurityLevel: UInt64` | The supported security level value. |
| `public var clientKextDenyListVersion: UInt32` | The version of the client's Kext deny list. |
| `public var deviceIdentity: FPSDeviceIndentity` | The identity information for the device interacting with the server. |
| `public var deviceInfo: FPSDeviceInfo` | The information about the device interacting with the server. |
| `public var numberOfSupportedKeyFormats: UInt32` | The number of key formats that the client supports. |
| `public var supportedKeyFormats: [UInt64]` | A list of supported key formats. |
| `public var vmDeviceInfo: Optional<VMDeviceInfo>` | Host and guest device information, if running in a virtual machine. |
| `public var ext: SPCDataExtension` | The additional extension data. |

### FPSServerCKCData

Swift

```
public struct FPSServerCKCData {
 public var ck: [UInt8]
 public var iv: [UInt8]
 public var r1: [UInt8]
 public var keyDuration: FPSServerKeyDuration
 public var hdcpTypeTLLVValue:
   base_constants.FPSHDCPRequirement

 public var contentKeyTLLVTag: UInt64
 public var contentKeyTLLVPayload: [UInt8]

 public var ext: CKCDataExtension
}
```

| API | Description |
|---|---|
| `public struct FPSServerCKCData { }` | Data to add to the CKC TLLVs. |
| `public var ck: [UInt8]` | An `Array` of the content key that encrypts or decrypts the content. |
| `public var iv: [UInt8]` | An `Array` of the initialization vector that initializes encryption. |
| `public var r1: [UInt8]` | An `Array` of the r1 values. |

| | |
|---|---|
| `public var keyDuration: FPSServerKeyDuration` | An instance of `FPSServerKeyDuration` and the duration that the key is valid. |
| `public var hdcpTypeTLLVValue: base_constants.FPSHDCPRequirement` | A value indicating the HDCP requirements. |
| `public var contentKeyTLLVTag: UInt64` | The tag associated with the content key. |
| `public var contentKeyTLLVPayload: [UInt8]` | The payload of the content key TLLV. |
| `public var ext: CKCDataExtension` | An extension that holds additional data. |

## `FPSDeviceIdentity`

Swift

```
public struct FPSDeviceIndentity {
  public var isDeviceIdentitySet: Bool
  public var fpdiVersion: UInt32
  public var deviceClass: UInt32
  public var vendorHash: [UInt8]
  public var productHash: [UInt8]
  public var fpVersionREE: UInt32
  public var fpVersionTEE: UInt32
  public var osVersion: UInt32
}
```

| API | Description |
|---|---|
| `public struct FPSDeviceIndentity { }` | Information to help identify the client device type, including vendor and product hashes, REE and TEE versions (only for third-party devices), and OS version (only for Apple products).<br><br>**Note:** Only devices running FairPlay client software released in 2021 or later send this TLLV. Prioritize using it over `FPSDeviceInfo` for client device type information. |
| `public var isDeviceIdentitySet: Bool` | A Boolean value that indicates whether the device identity is set. |
| `public var fpdiVersion: UInt32` | The current FPDI version. |
| `public var deviceClass: UInt32` | The device class for categorizing the type of device. |

| | |
|---|---|
| `public var vendorHash: [UInt8]` | A hash representing the vendor of the device. |
| `public var productHash: [UInt8]` | A hash representing the product of the device. |
| `public var fpVersionREE: UInt32` | The current version of the REE on the device, only applicable for third-party devices. |
| `public var fpVersionTEE: UInt32` | The current version of the TEE on the device, only applicable for third-party devices. |
| `public var osVersion: UInt32` | The current operating system version, only applicable for Apple products. |

### FPSDeviceInfo

Swift

```
public struct FPSDeviceInfo {
  public var isDeviceInfoSet: Bool
  public var deviceType: UInt64
  public var osVersion: UInt32
}
```

| API | Description |
|---|---|
| `public struct FPSDeviceInfo { }` | Basic information about the client device, including device type and OS version.<br><br>**Note:** This is a legacy TLLV. Use `FPSDeviceIdentity` instead, if available. |
| `public var isDeviceInfoSet: Bool` | A Boolean value that indicates whether the device info is set. |
| `public var deviceType: UInt64` | The client device type. |
| `public var osVersion: UInt32` | The current operating system version. |

### FPSServerTLLV

Swift

```swift
public struct FPSServerTLLV {
  public var tag: UInt64
  public var value: [UInt8]
}
```

| API | Description |
| --- | --- |
| `public struct FPSServerTLLV { }` | Contains the tag and value fields for a TLLV. |
| `public var tag: UInt64` | The tag identifying the specific field. |
| `public var value: [UInt8]` | An `Array` that stores the value associated with the tag. |

## FPSServerClientFeatures

Swift

```swift
public struct FPSServerClientFeatures {
  public var supportsOfflineKeyTLLV: Bool
  public var supportsOfflineKeyTLLVV2: Bool
  public var supportsSecurityLevelBaseline: Bool
  public var supportsSecurityLevelMain: Bool
  public var supportsHDCPTypeOne: Bool

  public var ext: ClientFeaturesExtension
}
```

| API | Description |
| --- | --- |
| `public struct FPSServerClientFeatures { }` | Contains fields that indicate whether the client supports certain features, including offline key V1 vs V2, Baseline vs Main security levels, and HDCP Type 1. |
| `public var supportsOfflineKeyTLLV: Bool` | A Boolean value that indicates whether the client supports the offline key TLLV format for version 1. |
| `public var supportsOfflineKeyTLLVV2: Bool` | A Boolean value that indicates whether the client supports the offline key TLLV format for version 2. |
| `public var supportsSecurityLevelBaseline: Bool` | A Boolean value that indicates whether the client supports the baseline security level. |
| `public var supportsSecurityLevelMain: Bool` | A Boolean value that indicates whether the client supports the main security level. |

| | |
|---|---|
| `public var supportsHDCPTypeOne: Bool` | A Boolean value that indicates whether the client supports HDCP Type 1. |
| `public var ext: ClientFeaturesExtension` | An extension that holds additional client features. |

### FPSServerSPCDataParser

| Swift | |
|---|---|
| `public struct FPSServerSPCDataParser { public var currentOffset: Int public var TLLVs: [FPSServerTLLV] public var parsedTagValues: [UInt64] }` | |
| API | Description |
| `public struct FPSServerSPCDataParser { }` | An intermediary data structure for parsing the SPC. It holds the current offset within the SPC data and parsed tags, along with the parsed TLLVs. |
| `public var currentOffset: Int` | The current offset within the SPC. |
| `public var TLLVs: [FPSServerTLLV]` | A current list of parsed TLLVs. |
| `public var parsedTagValues: [UInt64]` | A list of parsed tag values, where each value corresponds to a specific tag that the key server reads from the SPC data. |

### FPSServerMediaPlaybackState

| Swift | |
|---|---|
| `public struct FPSServerMediaPlaybackState { public var date: UInt32 public var playbackState: UInt32 public var playbackId: UInt64 }` | |
| API | Description |
| `public struct FPSServerMediaPlaybackState { }` | Contains information such as the date, playback state, and playback ID. |
| `public var date: UInt32` | The date associated with the playback state. |
| `public var playbackState: UInt32` | The current playback state of the media. |

| | |
|---|---|
| `public var playbackId: UInt64` | A unique identifier for the playback session. |

### FPSServerKeyDuration

Swift

```
public struct FPSServerKeyDuration {
  public var leaseDuration: UInt32
  public var rentalDuration: UInt32
  public var playbackDuration: UInt32
  public var keyType: UInt32

  public var ext: KeyDurationExtension
}
```

| API | Description |
|---|---|
| `public struct FPSServerKeyDuration { }` | Contains information about different key durations, including lease, rental, and playback duration, along with the key type. |
| `public var leaseDuration: UInt32` | The duration of the lease for the key, in seconds, before it expires. |
| `public var rentalDuration: UInt32` | The rental duration of the key. |
| `public var playbackDuration: UInt32` | The duration of content playback for the key. |
| `public var keyType: UInt32` | The type of key — whether it's for leasing, rental, or playback. |
| `public var ext: KeyDurationExtension` | Additional extended information related to key durations. |

`VMDeviceInfo`

| Swift |
| --- |
| `public struct VMDeviceInfo {`<br>`  public var hostDeviceClass: base_constants.FPSDeviceClass`<br>`  public var hostOSVersion: UInt32`<br>`  public var hostVMProtocolVersion: UInt32`<br>`  public var guestDeviceClass: base_constants.FPSDeviceClass`<br>`  public var guestOSVersion: UInt32`<br>`  public var guestVMProtocolVersion: UInt32`<br>`}` |

| API | Description |
| --- | --- |
| `public struct VMDeviceInfo {` | Contains information about the VM host and guest, if the requesting device is running in a VM. |
| `public var hostDeviceClass: base_constants.FPSDeviceClass` | Device class of the VM host. |
| `public var hostOSVersion: UInt32` | OS version of the VM host. |
| `public var hostVMProtocolVersion: UInt32` | FairPlay virtualization protocol version used by the VM host. |
| `public var guestDeviceClass: base_constants.FPSDeviceClass` | Device class of the VM guest. |
| `public var guestOSVersion: UInt32` | OS version of the VM guest. |
| `public var hostVMProtocolVersion: UInt32` | FairPlay virtualization protocol version used by the VM guest. |

## Parse input JSON files

JSON parsing begins by calling `parseRootFromString()` or `parseRootFromJson()`. These functions take in a string or a file, respectively, and convert it into the Swift internal JSON data structure. Then the program parses the JSON files into usable data structures. After ingesting the JSON files into the structure, the `processOperations()` and `parseOperations()` functions complete the parsing. These functions convert the data into a usable `FPSOperations` data structure, and decrypt and parse the SPC and its TLLVs. The following table describes the purpose of the functions for parsing input JSON files:

| Function | Description |
| --- | --- |
| `processOperations()` | The main function for handling a request. It takes in the ingested JSON file and returns the output JSON file. |
| `parseOperations()` | Handles parsing the input JSON file into the usable `FPSOperations` data structure. |

| | |
|---|---|
| `parseCreateCKCOperation()` | Parses a single `create-ckc` request into a usable `FPSOperation`.<br><br>Parses ID, SPC, check-in, and any asset info. |
| `parseAssetInfo()` | If there is asset info available, this function parses elements such as content key and IV, lease duration, and offline HLS. |

## Decrypt and parse SPC TLLVs

The `readNextTLLV()` function reads the TLLV data, and `parseTLLV()` parses it. First, `parseTLLV()` uses the TLLV tag to identify the specific TLLV to parse and then calls the parsing function for that TLLV (parsing functions each have their own file and are in the `parse_spc_TLLVs/` directory). If `parseTLLV()` doesn't find a matching TLLV tag, it drops the TLLV. There's a full list of TLLVs under the `FPSTLLVTagValue` enumeration. The following table describes the functions you use to decrypt and parse SPC TLLVs:

| Function | Description |
|---|---|
| `createResults()` | Sets the result ID and calls `genCKCWithCKAndIV()`. |
| `genCKCWithCKAndIV()` | Checks the SPC version, and calls functions to parse the SPC, query a database if needed, populate the `fpsResult` data structure, create the key payload, and generate the CKC. |
| `parseSPCV1()` | Calls functions to parse the SPC container, decrypt the SPC, parse the decrypted SPC, and check supported features. |
| `parseSPCContainer()` | Parses the unencrypted section of the SPC:<br>• Version<br>• 16-byte IV<br>• Encrypted AES wrapped key<br>• Certificate hash<br>• 4-byte SPC size |
| `decryptSPCData()` | Decrypts the AES wrapped key using RSA OAEP, and decrypts the SPC using the decrypted AES wrapped key. |
| `parseSPCData()` | Initializes play info, then loops through SPC, reading each TLLV and parsing it. It then saves the TLLVs in case the CKC needs to return them, and extracts TLLVs that the return tags specify.<br><br>**Note:** TLLV parsing exists under `parseTLLV()`. |

| | |
|---|---|
| `checkSupportedFeatures()` | Sets feature flags based on the parsed SPC information:<br>• `supportsLease`<br>• `supportsOfflineKeyTLLV`<br>• `supportsOfflineKeyTLLVV2`<br>• `supportsSecurityLevelBaseline`<br>• `supportsSecurityLevelMain`<br>• `supportsHDCPTypeOne` |
| `populateServerCtxResult()` | Copies needed SPC information into the results structure. |

## Create CKC TLLVs

The key server creates the content key within the `createContentkeyPayloadCustomImpl()` function in the extension. However, you need to call the `KSMCreateKeyPayload()` function from the object files in the prebuilt directory. After generating the content key payload, the program calls `generateCKCV1()` to create the CKC TLLVs. After parsing all of the `create-ckc` requests and creating the resulting CKCs, the key server serializes the final result JSON file with the `serializeResults()` function. The following table describes the functions you use to create CKC TLLVs:

| Function | Description |
| --- | --- |
| `serializeCKCData()` | Serializes TLLVs to add to the CKC:<br>• Content key<br>• R1<br>• Server return tags<br>• HDCP requirement<br>• Security level |
| `deriveAntiReplayKey()` | Derives the encryption key from the antireplay seed and R1. |
| `encryptCKCData()` | Encrypts the CKC using the antireplay key. |
| `serializeCKCContainer()` | Serializes the encrypted CKC, along with additional unencrypted fields:<br>• 4-byte version<br>• 4-byte reserved field<br>• 16-byte IV<br>• CKC data size<br>• CKC data |

### Input/Output JSON format

By default, the key server module requires asset information (content key, content IV, content type, HDCP requirement, and so forth.) to pass as part of the input JSON file. Alternatively, if asset information is unknown at the time of input JSON creation, you can implement the `queryDatabaseCustom()` function to look up asset information based on the `asset-id` inside the SPC. The following table provides the input JSON fields:

| Field name | Type | Description |
| --- | --- | --- |
| `fairplay-streaming-request` | Object | Contains the information for the FairPlay streaming request. |
| `version` | Integer | The input format version. This is always 1. |
| `create-ckc` | Object | Links inputs and outputs in the same request. |
| `id` | Integer | The value for correlating inputs and outputs if there are multiple in the same request. |

| Field name | Type | Description |
| --- | --- | --- |
| `spc` | String | A server playback context that the FairPlay library generates on the client device. |
| `check-in` | Boolean | `True` for sync operations. |
| `asset-info` | Object Array | Information about the requested asset. |
| `content-key` | String | Not required for lease renewals. |
| `content-iv` | String | Not required for lease renewals. |
| `content-type` | String | The type of content, such as `audio`, `hd`, and `uhd`. If not present, defaults to `unknown`. |
| `lease-duration` | Integer | The license duration, in seconds, starting from SPC creation time. Required for lease requests. Mutually exclusive with `offline-hls`. |
| `hdcp-type` | Integer | HDCP requirement:<br>–1 for no HDCP<br>`0` for HDCP Type 0<br>1 for HDCP Type 1.<br>If not present, the default is `0`. |
| `offline-hls` | Object | Required for persistent license requests. Mutually exclusive with `lease-duration`. |
| `rental-duration` | Integer | The rental duration, in seconds, starting from asset download time. |
| `playback-duration` | Integer | The lease duration, in seconds, starting from asset first playback time. |
| `stream-id` | String | The unique ID of each HLS substream. |
| `title-id` | String | The ID of a title (program). Same for all HLS substreams of a given title. |

Below is a sample input JSON file for a streaming request:

```
{
    "fairplay-streaming-request": {
        "version": 1,
        "create-ckc": [
            {
                "id": 1,
                "spc": “AAAAAgAAAACIiMWpQhMDI6pMnx2nfIiIMoaz9xQol...”,
                "asset-info": [
                    {
                        "content-key": "0102030405060708090A0B0C0D0E0F10",
                        "content-iv": "F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF",
                        "content-type": "hd",
                        "hdcp-type": 0,
                        "lease-duration": 1200
                    }
                ]
            }
        ]
    }
}
```

Below is a sample input JSON file for an offline request:

```
{
    "fairplay-streaming-request": {
        "version": 1,
        "create-ckc": [
            {
                "id": 1,
                "spc": “AAAAAQAAAACIiMWpQhMDI6pMnx2nfIiITwKbcTan4UepHXxB...”,
                "asset-info": [
                    {
                        "content-key": "ab07634237ab000fad0d2f29797c8f74",
                        "content-iv": "9a52030a2eb83b14f2e7989b8869c894",
                        "content-type": "uhd",
                        "hdcp-type": 1,
                        "offline-hls": {
                            "stream-id": "171062176140000000000000000000000",
                            "title-id": "5E785A7C000000000000000000000000",
                            "rental-duration": 2592000,
                            "playback-duration": 172800
                        }
                    }
                ]
            }
        ]
    }
}
```

The following table provides the output JSON fields:

| Field name | Type | Description |
| --- | --- | --- |
| `fairplay-streaming-response` | Object | Contains the information for the FairPlay streaming request. |
| `create-ckc` | Object | Links inputs and outputs in the same request. |
| `id` | Integer | The value for correlating inputs and outputs if there are multiple in the same request. |
| `status` | Integer | Returns a status. A value of `0` is success; other values are errors. |
| `hu` | String | An anonymized unique ID of the playback device as a hex string. |
| `ckc` | String | The output content key context as a Base64 string. It’s not present for sync operations. |

| Field name | Type | Description |
| --- | --- | --- |
| `check-in-server-challenge` | String | A unique challenge that the server generates. This is only for sync requests. |
| `check-in-flags` | String | Specifies a sync TLLV flag setting. This is only for sync requests. |
| `check-in-title-id` | String | The title ID in the `offline-hls` parameters of the input request as a hex string. This is only for sync requests. |
| `duration-left` | String | The duration until the expiration for rentals. This is only for sync requests. |
| `check-in-stream-id` | String Array | An array of content IDs that the system checks in as hex strings. The is only for sync requests. |
| `fpdi-version` | Integer | The device identity TLLV version. This is only present when receiving a device identity TLLV. |
| `device-class` | Integer | Specifies the device class, such as Apple Mobile. This is only present when receiving a device identity TLLV. |
| `vendor-hash` | String | A unique identifier for the device vendor as a hex string. This is only present when receiving a device identity TLLV. |
| `product-hash` | String | A unique identifier for a product as a hex string. This is only present when receiving a device identity TLLV. |
| `fps-ree-version` | String | The current version of FairPlay software running in `REE/userland` as a hex string. This is only present when receiving a device identity TLLV. |
| `fps-tee-version` | String | The current version of FairPlay software running in `TEE/kernel` as a hex string. This is only present when receiving a device identity TLLV. |
| `os-version` | String | The OS version as a hex string. This is only present when receiving a device identity TLLV. |

Below is an example of an output JSON file:


```
{
    "fairplay-streaming-response": {
        "create-ckc": [
            {
                "id": 1,
                "status": 0,
                "hu": "DB27C96B93D9218D50943F1498A8055E69993C18",
                "ckc": “AAAAAQAAAAA83lRbbGmLWMgKQJnAtLi8AAABoKqMgPh8l6CWq...”
                "fpdi-version": 1,
                "device-class": 2,
                "vendor-hash": "CA0D91584DE3468C",
                "product-hash": "9A77725DAE435607",
                "fps-ree-version": "00000000",
                "fps-tee-version": "00000000",
                "os-version": "00100000"
            }
        ]
    }
}
```


For additional examples, see the `Test_Inputs` folder.

## Revision history

| SDK Version | Date | Notes |
| --- | --- | --- |
| 5.0 | 2024-10-08 | New document that describes how to build, set up, and customize FairPlay Streaming Server SDK 5. |
| 5.1 | 2025-05-05 | Split Rust and Swift into separate documents.<br>Minor edits and formatting changes.<br>Added support for Linux ARM64.<br>Added sample http server using Vapor.<br>Added VM Device information. |

Apple Inc.
One Apple Park Way
Cupertino, CA 95014