Adobe® Access™

Version 4.0

April 2014

Using the Adobe Access Server for Protecting Content

Last updated 4/3/2014

Contents

Chapter 1: Introduction
Usage rules 1
Packaging Options 8

Chapter 2: Setting up the SDK
Setting up the development environment 11
Adobe Access credentials 12

Chapter 3: Working with policies
Creating a policy using the Java API 14
Updating a policy using the Java API 15
Policy criticality 16
Policy update lists 17

Chapter 4: Packaging media files
Encrypting content 19
Examining encrypted file content 20

Chapter 5: Pre-generating and embedding licenses
Pre-generating licenses 21
Embedding licenses 22

Chapter 6: Implementing the License Server
License Server deployment options 23
Processing Adobe Access requests 23
Handling Get Server Version requests 26
Handling Domain Registration requests 26
Handling Domain De-Registration requests 27
Handling authentication requests 28
Handling license requests 28
Handling synchronization requests 31
Handling FMRMS compatibility 32
Handling certificate updates 33
Performance tuning 34

Chapter 7: Revoking client credentials
Certificate Revocation Lists published by Adobe 35
Revoking DRM client and runtime credentials 35
Revoking machine credentials 36

Chapter 8: Creating video players

Chapter 1: Introduction

Adobe® Access™ is an advanced digital rights management and content protection solution for high-value audiovisual content. Using tools that you create using Java APIs, you can create policies, apply policies to files containing audio and video content, and encrypt those files. The high-level steps for performing these tasks are as follows:

1 Use the Java APIs to set the policy properties and encryption parameters.

2 Create a policy describing the usage roles for the content. (See "Working with policies" on page 14.)

You can create any number of policies. Most users create a small number of policies and apply them to many files.

3 Package a media file.

In this context, *packaging a file* means to encrypt it and apply a policy to it. (See "Packaging media files" on page 18.)

4 Implement the license server to issue a license to the user.

The encrypted content is now ready for deployment, and the client can request the license from the server.

The SDK provides a Java API to accomplish these tasks, and includes reference implementations of the license server, and command line tools that are based on the Java APIs. For information, see *Using the Adobe Access Reference Implementations*.

Usage rules

The following section describes the usage rules that you can specify in a policy.

What is new in Adobe Access 4.0

Adobe Access 4.0 supports the following new usage rules:

Authentication

User authentication

Specifies whether a credential, such as username and password, is required to acquire a license. If authenticated (identity-based) licensing is specified, the server authenticates the user before issuing a license.

Example use case: A subscription service might require a username/password to be entered before issuing a content license. A DVD or Blu-ray disc with Digital Copy might provide a code or other token as proof of payment, which can be redeemed for an electronic download.

Time-based rules

Adobe Access uses "soft enforcement" of time-based license restrictions. If a time right expires during playback of a video, the default behaviour of Adobe Access is to not restrict playback until the next time the video stream is recreated (by calling `Netstream.stop()` and `Netstream.play()`).

While soft enforcement is the default behavior, you can also enable hard enforcement by performing one of the following tasks:

- Have your Video Player periodically poll the license to make sure none of the time restrictions have expired. This can be accomplished by calling `DRMManager.loadVoucher(LOCAL_ONLY)`. An error code indicates that the locally-stored license is no longer valid.
- Whenever the user clicks the "pause" button, you can record the current video timestamp and then call `Netstream.stop()`. When the user clicks the "play" button, you can seek to the recorded location and then call "`Netstream.play()`".

Start date

Specifies the date after which a license is valid.

Example use case: Use an absolute date to issue content licenses ahead of the availability date of an asset, or to enforce an "embargo" period.

End date

Specifies the date after which a licenses expires.

Example use case: Use an absolute expiration date to reflect the end of distribution rights.

Relative end date

Specifies the license expiration date, expressed relative to the date of packaging.

Example use case: In an automated packaging process, use a single policy with this option for a series of videos to set the expiration date to 30 days relative to the date of packaging.

License caching duration

Specifies the duration a license can be cached on disk in the client's local License Store without requiring reacquisition from the license server. You can alternatively specify an absolute date/time after which a license can no longer be cached.

Once the cache expiration date has passed, the license is no longer valid, and the client must request a new license from the license server.

Example use case: Use the license caching duration to specify a fixed amount of time valid for a particular license, such as in a rental use case. A 30-day rental can be specified (with license caching) to indicate the total license duration within which to consume the content.

Playback window

Specifies the duration a license is valid after the first time it is used to play protected content.

Example use case: Some business models allow a rental period of 30 days but, once playback begins, it must be completed in 48 hours. This 48-hour longevity of the license is defined as the playback window.

Requirements for Synchronization

Specifies the frequency in which the client will synchronize its state with the server. If the client has been issued an out-of-band license (without a license server being contacted), usage rules may specify that the client must send synchronization messages to the server in order to synchronize the client's secure time and report client state to the server.

The synchronization behavior is defined using the following parameters:

- Start Interval — Specifies how long to wait after the last successful synchronization to start another synchronization request.
- Hard Stop Interval — (Optional). Disallow playback if a successful synchronization has not occurred in the specified amount of time.
- Force Sync Probability — (Optional). Probability with which the client should send a synchronize message before the next start interval.

Note: *This usage rule is supported by Adobe Access clients version 3.0 and higher. The behavior on older clients depends on the minimum client version supported by the license server. See, "Minimum Client Version" on page 30.*

Runtime and application restrictions

White-list for Adobe® Primetime applications allowed to play protected content

Specifies the Adobe Primetime applications that can play content. Specify the application ID, minimum version, maximum version, and publisher ID.

Example use case: Use this rule to limit playback to a particular application, or to control which version can access the content.

Note: *If you are using Adobe Flash Builder to build protected applications, make sure that you do not deploy the application in 'Debug' mode. When deploying an application in 'Debug' mode, the Flash Builder will append ".debug" to the AIR application ID. This will cause the white-list functionality in Adobe Access to behave unexpectedly.*

White-list for Adobe® Flash® Player SWFs allowed to play protected content

Specifies the SWF files that can play content. Specify the SWF file by a SWF URL or a SHA-256 digest computed using the contents of the SWF. If you use the SHA-256 digest, this usage rule also specifies the maximum amount of time to allow for the client to download and verify the SWF.

Example use case: Conceptually equivalent to SWF Verification in the case of Flash Media Server, but enforced on the client side to limit which video players can play the content. Note that Adobe Access behavior is different in regards to enforcement of the child SWF compared to the parent SWF.

Black-list of DRM Clients restricted from accessing protected content

Adobe Access DRM module versions restricted from accessing protected content.

Specifies the DRM client that cannot access content. Specified by DRM client version and platform.

Example use case: In the event of a security breach, a newer version of the DRM client can be specified as the minimum version required for license acquisition and content playback. The license server checks that the DRM client making the license request meets the version requirements before issuing a license. The DRM client also checks the DRM version in the license before playing content. This client check is required in the case of domains where a license may be transferred to another machine.

A DRM client version may be identified by the attributes specified in the following table:

Attribute	Supported Values	Match Criteria	Description
Environment	"PC", "PortingKit"	Exact Match	Identifies whether the client is running on a Desktop or any other device.
OS	"Win", "Mac", "Linux", "Android", "iOS"	Exact Match	Platform.
Architecture	"32", "64"	Exact Match	32-bit or 64-bit.
Screen Type	"PC", "Mobile", "TV"	Exact Match	
Release Version	A valid version number. For instance, "2.0.0".	Matches if client version is less than or equal to the specified version.	Version number is specified as a combination of numbers and periods (".") of any length.
OEM Vendor	OEM Vendor string.	Exact Match	OEM Vendor identification string for the device using the porting kit.
Model	Model string.	Exact Match	Device model identification string for the device using the porting kit.

Note: *When specifying an entry in the blacklist, values may be set for one or more of the attributes mentioned in the previous table. Any attribute that is not specified is treated as a wildcard. If the DRM client matches all the values specified in a blacklist entry, protected content may not be accessed by that client.*

Blacklist of application runtimes restricted from accessing protected content

Application Runtimes restricted from accessing protected content.

Specifies the version of the Primetime or Flash Runtime that cannot access content. Specify the restricted runtime (Flash Player, AIR, or iOS), platform, and version.

Example use case: Similar to the DRM Client blacklist, the latest version of the Flash Player, AIR, or iOS runtimes can be specified as the minimum version required for license acquisition and content playback.

The application runtime may be identified by any of the attributes supported for DRM client versions, in addition to the following attributes:

Attribute	Supported Values	Match Criteria	Description
Application	"FlashPlayer", "AIR", "iOS"	Exact Match	Identifies the name of the application runtime.

Minimum security level for DRM and runtimes

Specifies the security level required to access content. Specified by individual security levels for each component. The default security level for DRM/Runtime modules is 10000.

Example use case: Certain types of content (for example HD video) might require a higher security level than other types.

Device capabilities required to play protected content

Specifies the hardware capabilities required to access content. Information about the hardware capabilities is available for devices that use the porting kit.

The device capabilities may be identified by the attributes specified in the following table:

Attribute	Supported Values	Match Criteria	Description
Non-user accessible bus	"true" or "false"	Exact Match	If true, device must not have a user accessible bus.
Hardware root of trust	"true" or "false"	Exact Macth	If true, device must have a hardware root of trust.

Note: *This usage rule is supported by Adobe Access clients version 2.0.2 and higher. The behavior on older clients depends on the minimum client version supported by the license server. See, "Minimum Client Version" on page 30.*

Jailbreak Enforcement

On platforms that supports jailbreak detection (such as Adobe Primetime on iOS), enabling jailbreak enforcement disallows playback of content if jailbreak has been detected on the device.

Note: *This usage rule is supported by Adobe Access clients version 4.0 and higher, but has no effect on platforms that do not support jailbreak detection. The behavior on older clients depends on the minimum client version supported by the license server.*

Other policy options

Custom usage rules

Specifies custom usage rules. Custom data can be included in licenses issued by the License Server. The interpretation/handling of this data is completely up to the implementation of the client application and license server.

Example use case: Enables extensibility of usage rules by allowing other business rules to be conveyed securely as part of the policy and/or content license. For security reasons, because these usage rules are enforced in custom client application code, this option should be used in conjunction with the AIR application or Flash Player SWF white-list options. For more information, see ""Runtime and application restrictions" on page 3".

Enhanced license chaining

Allows a license to be updated using a parent root license for batch updating of licenses.

Adobe Access 2.0 supported license chaining in which both the leaf and root licenses are bound to a specific machine. Adobe Access 3.0, has support for enhanced license chaining, in which a leaf is bound to a root license, and only the root license is bound to a specific machine or domain. The enhanced license chaining supports embedding leaf license with the content, and the client only needs to acquire the root license from the license server in order to consume the protected content.

To enable the enhanced licence chaining, a root encryption key is assigned to the policy. The root encryption key is used to cryptographically bind the leaf license to the root license.

Note: *The enhanced license chaining is supported by Adobe Access clients version 3.0 and higher. If an older client requests a license for content that supports the enhanced license chaining, the license server can still issue a license to this client using license chaining supported by Adobe Access 2.0.*

Example use case: Use this option to update any linked licenses by downloading a single root license. For example, implement subscription models where content can be played back as long as the user renews the subscription on a monthly basis. The benefit of this approach is that users only have to do a single license acquisition to update all of their subscription licenses.

Multiple play rights

Allows different usage rules to be specified for different platforms or applications.

Example use case: Using multiple play rights, you can create a policy to specify that Output Protection is required on the Microsoft® Windows® platform, and is optional on the Apple® Macintosh® and Linux® platforms.

Remote Key Delivery

Adobe Primetime on iOS supports two options for key delivery to iOS clients:

- Local
- Remote

The remote key delivery is enabled through the policy used to package content (changing this setting requires repackaging of content), When remote key delivery is enabled, an Adobe Access Key Server must be deployed to handle key requests from iOS clients, but there is no change to the workflow for clients on other platforms.

Note: *The Key delivery selection only impacts iOS clients.*

For information, see *Using the Adobe Access Key Server*.

Domain Registration

As an alternative to binding a license to a specific device, Adobe Access 3.0 supports binding licenses to a domain. Multiple devices may join a domain and receive domain tokens. After a device in the domain acquires a license, the license may be transferred to any other device in the domain, and those devices can play the content without acquiring a license directly from the license server.

To support the domain-bound licenses, the policy must specify the domain server with which the client must register. The policy will also specify the authentication requirements for the domain server (whether anonymous access is allowed or whether the server requires username/password or custom authentication).

Domain registration and domain-bound licenses are supported by Adobe Access clients version 3.0 and higher. If an older client or a Adobe Access 3.0 client in Flash Player requests a license for content that supports domain registration, the license server may issue a license using an alternative policy that supports binding to a specific device.

Output protection

Output protection controls

Control whether output to external rendering devices is protected. Specify analog and digital outputs independently.

Controls whether output to external rendering devices should be restricted. An external device is defined as any video or audio device that is not embedded in the computer. The list of external devices excludes integrated displays, such as in notebook computers. Analog and digital output restrictions can be specified independently.

The following options/levels of enforcement are available:

Option	Supported in analog devices	Supported in digital devices
Required — Analog Copy Protection (ACP) or Copy Generation Management System - Analog (CGMS-A) output protection must be enabled in order to play content to an external device. Adobe Access clients must enable output protection using ACP or CGMS-A. On devices that support both, the Adobe Access 3.0 clients will attempt to enable both. However, only one must be enabled to play the content.	YES	YES
ACP Required — ACP output protection is required. Playback is not allowed on CGMS-A. Adobe Access 2.0 clients do not support this option. If set, an Adobe Access 2.0 client will behave as if the "No Playback" option was specified.	YES	-
CGMS-A Required — CGMS-A output protection is required. Playback is not allowed on ACP. Adobe Access 2.0 clients do not support this option. If set, an Adobe Access 2.0 client will behave as if the "No Playback" option was specified.	YES	-
Use if available — Attempt to enable ACP and CGMS-A output protection if available and allow playback if not available. Adobe Access 3.0 clients will attempt to enable both ACP and CGMS-A, if possible. Adobe Access 2.0 clients will only attempt to enable either ACP or CGMS-A. For instance, an attempt will be made by the Adobe Access client to enable either ACP or CGMS-A. If the attempt succeeds, the other option will not be enabled. If the attempt fails, a second attempt will be made to enable the other option. Even if both the attempts fail, the content will be played anyway.	YES	YES
Use ACP if available — Attempt to enable ACP output protection if available, but allow playback if not available. Protection is not available on CGMS-A. Adobe Access 2.0 clients do not support this option. If set, an Adobe Access 2.0 client will behave as if the "No Protection" option was specified.	YES	-
Use CGMS-A if available — Attempt to enable CGMS-A output protection if available, but allow playback if not available. Protection is not available on ACP. Adobe Access 2.0 clients do not support this option. If set, an Adobe Access 2.0 client will behave as if the "No Protection" option was specified.	YES	-
No protection — No output protection enablement is enforced for analog and digital outputs.	YES	YES
No playback —Do not allow playback to an external device for analog and digital outputs.	YES	YES

Note: *While these rules are consistently enforced across all platforms, currently it is only possible to securely turn on output protection on Windows platforms. On other platforms (such as Macintosh and Linux) there are no supporting operating system functions available to third party applications.*

Example use case: Some content might enforce output protection controls, and the level of protection can be set by the content distributor. If "Required" is specified and playback is attempted on a Macintosh, the client does not play back content on external devices. The content will, however, play back on internal monitors.

If "Required" is specified and playback is attempted on Linux, the client does not play back content on any devices because it is not possible to differentiate between internal and external devices.

If you specify "Use if available", output protection is turned on where possible. For example, on Windows machines that support the Certified Output Protection Protocol (COPP), the content is passed with output protection to an external display. This example is sometimes known as "selectable output control".

Packaging Options

The following encryption options are selected at packaging time and cannot be modified during license acquisition.

Key Rotation

Supported only in Adobe Access Professional.

During packaging, typically the content is encrypted using the Content Encryption Key (CEK), and the client obtains a license containing the CEK in order to consume the content. When key rotation is enabled, the Rotation Key is used to encrypt the content, and the key can be changed so that each Rotation Key is only used to encrypt a portion of the content. The Rotation Keys are protected using the Content Encryption Key, and the client still obtains a single license containing the CEK in order to consume the content. The packager implementation can control the Content Encryption Key and Rotation Keys used, as well as the frequency with which the Rotation Keys change.

Content packaged using key rotation can only be played on Adobe Access clients version 3.0 and higher. Older clients would need to upgrade in order to play this content.

Out-of-band Licenses

Supported only in Adobe Access Professional.

Using Adobe Access 3.0 Professional, it is possible to implement a workflow in which the clients obtain pre-generated licenses out-of-band, eliminating the need to deploy a license server. To support this workflow, an option indicating that no license server is available should be specified at packaging time. This will prevent the Adobe Access client from attempting to request a license for this content from a license server.

Content which indicates the no license server is available, can only be played on Adobe Access clients version 3.0 and higher. Older clients would need to upgrade in order to play this content.

Encrypting tracks

Specifies which parts of the content are encrypted: audio, video, or both.

Example use case: Permits encrypting just the tracks of the content that require protection, thus reducing the decryption overhead on the client and improving playback performance.

Encrypting script data

Specifies whether script data embedded in the content is encrypted.

Note: *This rule only applies for FLV file format. Script data is always left in the clear for files in the F4V format.*

Example use case: Use this option to leave script data unencrypted, which allows for metadata aggregation tools to read the metadata of protected content.

Partial encryption level

Specifies whether all frames, or only a subset of frames, should be encrypted. There are three levels of encryption: low, medium and high.

Note: *For video track in F4V/H.264 files only.*

Partial encryption is designed to give content providers granularity to encode the content in parts. Encryption of content adds CPU overhead to the device that is decrypting and viewing the content. Use partial encryption to reduce CPU overhead while maintaining very strong protection of the content. A motivating case for using this feature is a single piece of content that is intended to be playable across low, medium, and high powered devices.

Due to the nature of video encoding, it is not necessary to encrypt 100% of video to make it unplayable if it is stolen. Partial encryption has three settings, low, medium, and high, and the associated percentages of encryption are dependent upon how the video is encoded. Because of this encoding dependency, the percentage of your content that is encrypted falls within the following ranges:

- High: Encrypts all samples.
- Medium: Encrypts a target 50% of the data.
- Low: Encrypts a target 20 to 30% of the data.

These settings were designed with the following rule: Any content that is encrypted at the low setting is also encrypted at the medium setting. This ensures that the same piece of content distributed at low encryption by one party and distributed at medium encryption by another party does not compromise the protection of the content.

Example use case: Reducing the encryption level decreases the decryption overhead on the client and improves playback performance on low-end machines.

Initial portion of content in the clear

Specifies an optional amount of time, in seconds, that the beginning of the content is left in the clear (meaning it is not encrypted).

Example use case: Allows for faster time to playback while the Adobe Access client downloads the license in the background. The unencrypted portion of the video begins playback immediately, while the Adobe Access initialization and license acquisition occur behind the scenes. With this feature turned off, users may notice a delay in playback experience, as the client machine is performing all of the licensing steps before any video playback occurs.

Custom metadata

Specify custom key/value to add to content metadata that can be interpreted by the server application.

The Adobe Access content metadata format allows for inclusion of custom key/value pairs at packaging time to be processed by the license server during license issuance. This metadata is separate from the policy and can be unique for each piece of content.

Example use case: During a Beta phase, you include the custom property "Release:BETA" at packaging time. License servers can vend licenses to this content during the Beta period, but after the Beta period expires, the license servers disallow access to the content.

Multiple policies

Specify multiple policies to be associated with a single piece of content. The specific policy to be used is determined by the license server.

Example use case: If the same asset is used for both electronic sell-through and rental models, this option allows you to specify different sets of usage rules for each business model. The license server can choose which policy to use based on whether the customer purchased or rented the content.

Chapter 2: Setting up the SDK

To set up the Adobe® Access™ for use, copy files from the DVD. These files include JAR files containing code, certificates, and third-party classes. In addition, request a certificate from Adobe Systems Incorporated. You will be issued multiple credentials used to protect the integrity of packaged content, licenses, and communication between the client and server.

The Adobe Access 4.0 SDK is available in two types:

- Adobe Access Core SDK
- Adobe Access Professional SDK

The following table shows a basic comparison of Adobe Access SDKs:

Feature	Adobe Access Core SDK	Adobe Access Professional SDK
Flash Access 2.0 features	Available	Available
Key Rotation	-	Available
Domain Support	Available	Available
Enhanced License Chaining	Available	Available
Synchronization Messages	Available	Available
License Pre-Generation	-	Available
Embedded Licenses	-	Available

Setting up the development environment

From the DVD, copy the following SDK files for use in your development environment and your Java classpath:

- adobe-flashaccess-certs.jar (contains Adobe root certificates)
- adobe-flashaccess-sdk.jar (contains Adobe Access Core SDK classes)
- adobe-flashaccess-sdk-pro.jar (contains Adobe Access Professional SDK classes, required only for Professional features)

You need the following third party JAR files also located on the DVD in the SDK's "thirdparty" folder:

- bcmail-jdk15-141.jar
- bcprov-jdk15-141.jar
- commons-discovery-0.4.jar
- commons-logging-1.1.1.jar
- cryptoj.jar
- jaxb-api.jar
- jaxb-impl.jar
- jaxb-libs.jar

- relaxngDatatype.jar
- rm-pdrl.jar
- xsdlib.jar

For improved performance, you can optionally enable native support for cryptographic operations by deploying the platform-specific libraries located in the "thirdparty/cryptoj" folder of the SDK. To enable native support, add the library for your platform (jsafe.dll for Windows or libjsafe.so for Linux) to the path. The 32-bit and 64-bit versions of these libaries are provided. (Note that the 64-bit version should only be used if you have a 64-bit OS and you are running the 64-bit version of Java).

Additionally, an optional part of the SDK is adobe-flashaccess-lcrm.jar. This file is only needed for functionality related to Adobe Flash Media Rights Management Server (FMRMS) 1.x compatibility. If you previously deployed FMRMS 1.x and do not want to re-package your FMRMS-protected content, you must add support to your license server so it will be able to handle old content and clients.

Adobe Access credentials

There are 4 types of credentials required to use Adobe Access:

- **Packager**: used during packaging to sign the metadata added to the encrypted content
- **License Server**: used to protect the content encryption key in the metadata, and used by the license server to sign licenses
- **Transport:** used to protect requests/responses exchanged between the client and the license server
- **Domain CA**: used to issue domain certificates to devices that join a domain.

Requesting certificates

Enrollment is the process of requesting a certificate from Adobe. You can generate your keys and create a request that is sent to Adobe. Adobe then generates the certificate and sends it back to you. Adobe will not know the contents of the private key. Therefore, you must have a way to back up the key so that you can recover it in case of hardware failure.

Unlike the License Server, Packager or Transport certificate, the Domain CA certificate is not issued by Adobe. You can obtain this certificate from a Certificate Authority, or you can generate a self-signed certificate.

For instructions on how to obtain the Adobe Access credentials, see the *Adobe Access Certificate Enrollment Guide*.

Storing credentials

The SDK supports multiple ways of storing credentials (a public key certificate and its associated private key) , including on an HSM or as a PKCS12 file. Credentials are used when the private key is required (for example, for the packager to sign the metadata, or for the License Server to decrypt data encrypted with the License Server or Transport public key). Private keys must be closely guarded to ensure the security of your content and License Server. PKCS12 is a standard format for a file containing a credential encrypted using a password. The file extension .pfx is commonly used for files of this format.

Note: *Adobe recommends using an HSM for maximum security. For more information, see The Adobe Access Secure Deployment Guidelines.*

You can keep a private key on a Hardware Security Module (HSM) and use the SDK to pass in the credential you obtain from the HSM. To use a credential stored on an HSM, use a JCE provider that can communicate with an HSM to get a handle to the private key. Then, pass the private key handle, provider name, and certificate containing the public key to `ServerCredentialFactory.getServerCredential()`.

The SunPKCS11 provider is one example of a JCE provider which can be used to access a private key on an HSM (see the Sun Java documentation for instructions on using this provider). Some HSMs also come with a Java SDK which includes a JCE provider.

PEM and DER are two ways of encoding a public key certificate. PEM is a base-64 encoding and DER is a binary encoding. Certificate files typically use the extension .cer, .pem. or .der. Certificates are used in places where only the public key is required. If a component requires only the public key to operate, it is better to provide that component with the certificate instead of a credential or PKCS12 file.

Chapter 3: Working with policies

Using Adobe® Access™, content providers can apply policies to FLV files and F4V files. Using the policy management APIs, administrators can create, view details of, and update policies.

A *policy* defines how users can view content; it is a collection of information that includes security settings, authentication requirements, and usage rights. When policies are applied, encryption and signing allow content providers to maintain control of their content no matter how widely it is distributed. Protected files can be delivered by using Adobe® Flash® Media Server or an HTTP server. They can be downloaded and played in custom players built with Adobe® AIR®, Adobe® Flash® Player and Adobe® Primetime SDK for iOS. The policy is a template for the license server to use when it generates a license. The client may also refer to the policy before requesting a license to determine whether it needs to prompt the user to authenticate before issuing a license request to the server.

A policy specifies one or more rights that are granted to the client. Typically, a policy includes, at a minimum, the "Play Right". It is also possible to specify multiple Play Rights, each with different restrictions. When the client encounters a license with multiple Play Rights, it uses the first one for which it meets all the restrictions. For example, this feature can be used to enforce different output protection settings on different platforms. For sample code illustrating this example, see `CreatePolicyWithOutputProtection.java` in the Reference Implementation Command Line Tools "samples" directory.

You can accomplish the following tasks by using the policy management APIs:

- Create and update policies
- View policy details
- Manage policy update lists

For details about the Java API discussed in this chapter, see *Adobe Access API Reference*.

For information about the Policy Manager reference implementation, see *Using the Adobe Access Reference Implementations*.

Creating a policy using the Java API

To create a policy by using the Java API, perform the following steps:

1. Set up your development environment and include all of the JAR files mentioned in "Setting up the development environment" on page 11 within your project.
2. Create a `com.adobe.flashaccess.sdk.policy.Policy` object and specify its properties, such as the rights, license caching duration, and policy end date.

```
// Create a new Policy object.
// False indicates the policy does not use license chaining.
Policy policy = new Policy(false);

policy.setName("DemoPolicy");

// Specify that the policy requires authentication to obtain a license.
policy.setLicenseServerInfo
    (new LicenseServerInfo(AuthenticationType.UsernamePassword));

// A policy must have at least one Right, typically the play right
PlayRight play = new PlayRight();

// Users may only view content for 24 hours.
play.setPlaybackWindow(24L * 60 * 60);

// Add the play right to the policy.
List<Right> rightsList = new ArrayList<Right>();
rightsList.add(play);
policy.setRights(rightsList);

// Licenses may be stored on the client for 7 days after downloading
policy.setLicenseCachingDuration(7L * 24 * 60 * 60);
try {
    // Content will expire December 31, 2010
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    policy.setPolicyEndDate(dateFormat.parse("2010-12-31"));
} catch (ParseException e) {
    // Invalid date specified in dateFormat.parse()
    e.printStackTrace();
}
```

3 Serialize the `Policy` object and store it in a file or database.

```
// Serialize the policy
byte[] policyBytes = policy.getBytes();
System.out.println("Created policy with ID: " + policy.getId());

// Write the policy to a file.
// Alternatively, the policy may be stored in a database.
FileOutputStream out = new FileOutputStream("demopolicy.pol");
out.write(policyBytes);
out.close();
```

For the full source of this sample code, see *com.adobe.flashaccess.samples.policy.CreatePolicy* in the Reference Implementation Command Line Tools “samples” directory.

Updating a policy using the Java API

To update a policy by using the Java API, perform the following steps:

1 Set up your development environment and include all of the JAR files mentioned in “Setting up the development environment” on page 11 within your project.

2 Create a `Policy` instance and read in the policy from a file or database.

```
Policy policy = new Policy(policyBytes);
```

3 Update the `Policy` object by setting its properties, such as its name and usage rules.

```
// Change the policy name.
policy.setName("UpdatedDemoPolicy");

// Add DRM module restrictions to the play right
for (Right r: policy.getRights()) {
    if (r instanceof PlayRight) {
        PlayRight pr = (PlayRight) r;
        // Disallow Linux versions up to and including 1.9.  Allow
        // all other OSes and Linux versions above 1.9
        VersionInfo toExclude = new VersionInfo();
        toExclude.setOS("Linux");
        toExclude.setReleaseVersion("1.9");
        Collection<VersionInfo> exclusions = new ArrayList<VersionInfo>();
        exclusions.add(toExclude);
        ModuleRequirements drmRestrictions = new ModuleRequirements();
        drmRestrictions.setExcludedVersions(exclusions);
        pr.setDRMModuleRequirements(drmRestrictions);
        break;
    }
}
```

4 Serialize the updated `Policy` object and store it in a file or database.

```
// Serialize the policy.
byte[] policyBytes = policy.getBytes();
System.out.println("New policy revision number: "
    +   policy.getRevision());
// Write the policy to a file.
// Alternatively, the policy may be stored in a database.
FileOutputStream out = new FileOutputStream("demopolicy-updated.pol");
out.write(policyBytes);
out.close();
```

For the full source of this sample code, see `com.adobe.flashaccess.samples.policy.UpdatePolicy` in the Reference Implementation Command Line Tools "samples" directory.

Policy criticality

If new usage rules are used in the policies and these policies are used in content packaged for older license servers (which do not understand the new usage rules), you may specify how older license servers should behave. By default, the policy criticality is “true”, meaning that the license server must understand all parts of the policy in order to generate a license using the policy. If the policy criticality is set to “false”, an older license server may ignore parts of the policy it does not understand, and licenses generated by the server will not contain the new usage rules.

Adobe Access servers using version 2.0.2 of the SDK and higher will honor the policy criticality setting.

Policy update lists

If you update the usage rules in a policy after packaging the content, the license server needs to have the latest version in order to issue licenses using the updated policy. One way to achieve this is through a policy update list. A policy update list is a file containing a list of updated or revoked policies. When a policy is updated, generate a new Policy Update List and periodically push the list out to all the license servers.

For more information creating and updating policies, see *Working with policies*.

Working with Policy Update Lists

For license servers that do not have access to a database for storing information about policies, you may wish to use a policy update list to notify the license server of updated policies. Policy Update Lists may contain updated versions of policies or a list of policy IDs that have been revoked. If a policy update list is supplied in `HandlerConfiguration`, the SDK will enforce this list when issuing a license.

Policies may also be revoked if content owners or distributors want to discontinue issuing licenses under a particular policy. A policy update list may be used to enforce policy revocation in the SDK. Policy update lists may also be used to provide a list of updated policies to the SDK. Note that revoking a policy does not revoke licenses that have already been issued. It only prevents additional licenses from being issued under that policy.

Working with policy update lists involves the use of a `PolicyUpdateListFactory` object. To create a policy update list, load an existing policy update list, and check whether a policy has been updated or revoked by using the Java API, perform the following steps:

1. Set up your development environment and include all of the JAR files mentioned in "Setting up the development environment" on page 11 within your project.
2. Create a `ServerCredentialFactory` instance to load the credentials needed for signing.
3. Create a `PolicyUpdateListFactory` instance using the `ServerCredential` you created.
4. Specify the policy ID to be revoked.
5. Create a `PolicyRevocationEntry` object using the policy ID `String` you just created, and add it to the policy update list by passing it into `PolicyUpdateListFactory.addRevocationEntry()`. Generate the new policy update list by calling `PolicyUpdateListFactory.generatePolicyUpdateList()`. Similarly, updated policies can be added to the list using `PolicyUpdateEntry`.
6. If a policy update list already exists, you can serialize it for loading by calling `PolicyUpdateList.getBytes()`. To load the list, call `PolicyUpdateListFactory.loadPolicyUpdateList()` and pass in the serialized list.
7. Verify the signature is valid and the list was signed by the correct license server certificate by calling `PolicyUpdateList.verifySignature()`.
8. To check whether an entry was revoked, pass the policy ID `String` into `PolicyUpdateList.isRevoked()`. Alternatively, the list can be passed into `HandlerConfiguration` and it will be enforced when licenses are issued.

To add additional entries to an existing `PolicyUpdateList`, load an existing policy update list. Create a new `PolicyUpdateListFactory` instance. Call `PolicyUpdateListFactory.addEntries` to add all the entries from the old list to the new list. Call `PolicyUpdateListFactory.addRevocationEntry` or `addUpdatedEntry` to add any new revocation or update entries to the PolicyUpdateList.

For sample code demonstrating how to create a policy update list, load an existing policy update list, and check whether a policy has been revoked, see `com.adobe.flashaccess.samples.policyupdatelist.CreatePolicyUpdateList` in the Reference Implementation Command Line Tools "samples" directory.

Chapter 4: Packaging media files

Packaging refers to the process of encrypting and applying a policy to FLV or F4V files. Use the media packaging APIs to package files.

Packaging is de-coupled from the license server. There is no need for the packager to connect to the license server to exchange any information about the content. Everything the license server needs to know to issue the license is included in the content metadata.

When a file is encrypted, its contents cannot be parsed without the appropriate license. Adobe Access allows you to select which parts of the file to encrypt. Because Adobe® Access™ can parse the file format of the FLV and F4V content, it can intelligently encrypt selective parts of the file instead of the entire file as a whole. Data such as metadata and cue points can remain unencrypted so that search engines can still search the file.

It is possible for a given piece of content to have multiple policies. This could be useful, for example, if you would like to license content under different business models without having to package the content multiple times. For example, you could allow anonymous access for a short period of time, and after that allow the customer to buy the content and have unlimited access. If content is packaged using multiple policies, the License Server must implement logic for selecting which policy to use to issue a license.

Note: *The architecture allows for usage policies to be specified and bound to content when the content is packaged. Before a client can play back content, the client must acquire a license for that computer. The license specifies the usage rules that are enforced and provides the key used to decrypt the content. The policy is a template for generating the license, but the license server may choose to override the usage rules when issuing the license. Note that the license may be rendered invalid by such constraints as expiration times or playback windows.*

There are numerous options available when packaging content. These are specified in the `DRMParameters` interface and the classes implementing that interface, which are the `F4VDRMParameters` and `FLVDRMParameters`. With these classes you can set signature and key parameters, as well as indicate whether to encrypt audio content, video content, or script data. To see how these are implemented in the reference implementation, see the descriptions of the Media Packager command line options discussed in *Using the Adobe Access Reference Implementations*. These options are based on the Java API and are therefore available for programmatic usage.

The packaging options include:

- Encryption options (audio, video, partial encryption).
- License server URL (the client uses this as the base URL for all requests sent to the license server)
- License server transport certificate
- License sever certificate, used to encrypt the CEK.
- Packager credential for signing metadata

Adobe Access provides an API for passing in the CEK. If no CEK is specified, the SDK randomly generates it. Typically you need a different CEK for each piece of content. However, in Dynamic Streaming, you would likely use the same CEK for all the files for that content, so the user only needs a single license and can seamlessly transition from one bit rate to another. To use the same key and license for multiple pieces of content, pass the same `DRMParameters` object to `MediaEncrypter.encryptContent()`, or pass in the CEK using `V2KeyParameters.setContentEncryptionKey()`. To use a different key and license for each piece of content, create a new `DRMParameters` instance for each file.

When packaging content using key rotation, you can control the Rotation Keys used and how often the keys change. `F4VDRMParameters` and `FLVDRMParameters` implement the `KeyRotationParameters` interface. Through this interface, you can enable key rotation. You also need to specify a `RotatingContentEncryptionKeyProvider`. For each sample encrypted, this class determines the Rotation Key to use. You may implement your own provider, or use the `TimeBasedKeyProvider` included with the SDK. This implementation randomly generates a new key after a specified number of seconds.

In some cases you may need to store the content metadata as a separate file and make it available to the client separate from the content. To do this, invoke `MediaEncrypter.encryptContent()`, which returns a `MediaEncrypterResult` object. Call `MediaEncrypterResult.getKeyInfo()` and cast the result to `V2KeyStatus`. Then retrieve the content metadata and store it in a file.

All of these tasks can be accomplished using the Java API. For details about the Java API discussed in this chapter, see *Adobe Access API Reference*.

For information about the Media Packager reference implementation, see *Using the Adobe Access Reference Implementations*.

Encrypting content

Encrypting FLV and F4V content involves the use of a `MediaEncrypter` object. You can also package FLV and F4V files that contain only audio tracks. When encrypting H.264 content for lower-end devices, it may be practical to apply only partial encryption to improve performance. In such cases, an F4V file may be partially encrypted using the `F4VDRMParameters.setVideoEncryptionLevel` method.

To encrypt an FLV or an F4V file by using the Java API, perform the following steps:

1. Set up your development environment and include all of the JAR files mentioned in Setting up the development environment within your project.
2. Create a `ServerCredential` instance to load the credentials needed for signing.
3. Create a `MediaEncrypter` instance. Use a `MediaEncryperFactory` if you do not know what type of file you have. Otherwise you can create an `FLVEncrypter` or `F4VEncrypter` directly.
4. Specify the encryption options by using a `DRMParameters` object.
5. Set the signature options using a `SignatureParameters` object and pass the `ServerCredential` instance to its `setServerCredentials` method.
6. Set the key and license information using an `V2KeyParameters` object. Set the policies using the `setPolicies` method. Set the information needed by the client to contact the license server by calling the `setLicenseServerUrl` and `setLicenseServerTransportCertificate` methods. Set the CEK encryption options using the `setKeyProtectionOptions` method, and its custom properties using the `setCustomProperties` method. Finally, depending on the type of encryption used, cast the `DRMKeyParameters` object to one of `VideoDRMParameters`, `AudioDRMParameters`, `FLVDRMParameters`, or `F4VDRMParameters`, and set the encryption options.
7. Encrypt the content by passing the input and output files and encryption options to the `MediaEncrypter.encryptContent` method.

For sample code demonstrating how to encrypt content, see `com.adobe.flashaccess.samples.mediapackager.EncryptContent` in the Reference Implementation Command Line Tools “samples” directory.

Examining encrypted file content

To examine the contents of an FLV or an F4V file by using the Java API, perform the following steps:

1. Set up your development environment and include all of the JAR files mentioned in "Setting up the development environment" on page 11 within your project.
2. Create a `MediaEncrypter` instance.
3. Pass the encrypted file to the `MediaEncrypter.examineEncryptedContent` method, which returns a `KeyMetaData` object.
4. Inspect the information within the `KeyMetaData` object.

For sample code demonstrating how to extract DRM metadata from an encrypted file, see `com.adobe.flashaccess.samples.mediapackager.ExamineContent` in the Reference Implementation Command Line Tools "samples" directory.

Chapter 5: Pre-generating and embedding licenses

Using Adobe Access 4.0 Professional, it is possible to pre-generate licenses and embed licenses in content. This feature can be combined with the Enhanced License Chaining such that a Leaf license is pre-generated and embedded in the content, and the client can request a Root license (bound to a machine or domain) from a license server. Alternatively, client applications can implement a workflow where the device pre-registers with a server, the server pre-generates licenses bound to that device, and the client retrieves its licenses from a simple HTTP web server.

Pre-generating licenses

To pre-generate licenses, use `com.adobe.flashaccess.sdk.license.pregen.LicenseFactory.getInstance()` to obtain an instance of `LicenseFactory`. A License Server credential must be specified in order to sign the licenses generated by this factory. This class supports generating Leaf licenses without license chaining and Leaf and Root licenses with the "Enhanced license chaining" on page 5.

When generating a Leaf license, the content metadata must be specified using `initContentInfo()`. If the metadata includes multiple policies, or if you want to use a policy that was not in the metadata, use `setSelectedPolicy()` to specify the policy to use to generate the license. If you use a Policy Update List to track updates to policies, you can provide the Policy Update List to the License Factory before initializing the metadata using `setPolicyUpdateList()`.

When generating a Root license, the content metadata may be specified as described above. Alternatively, a Root license can be generated using a policy (`setSelectedPolicy()`) and license server URL (`setLicenseServerURL()`) instead of the metadata.

Note: *A License Server URL is required even if there is no Adobe Access License Server from which the clients can request a license. In this case, the License Server URL should specify a URL identifying the license issuer.*

If the policy uses Enhanced License Chaining, a License Server credential must be specified in order to decrypt the Root Encryption Key in the policy (`setRootKeyRetrievalInfo()`).

If the policy requires a domain bound license, use `setDomainCAs()` to specify the Domain issuers from which the license server will accept domain tokens. One or more Domain CA certificates must be provided in order to validate the license recipient.

If the policy requires remote key delivery for iOS devices, the Key Server Certificate must be provided using `setKeyServerCertificate()`, unless a chained Leaf is being generated.

To generate a license, invoke `generateLicense()` and specify the license type (Leaf or Root) and one or more recipient certificates. The recipient certificate will either be a machine certificate or domain certificate, depending on the requirements specified in the policy. If you are generating a chained Leaf, a recipient is not required. After the license has been generated, it is possible to override the usage rules that were specified in the policy. Finally, invoke `signLicense()` to sign the license and obtain an instance of `PreGeneratedLicense`. The license can now be saved to a file (use `getBytes()` to retrieve the serialized license) or embedded in encrypted content. See, Embedding Licenses.

For sample code demonstrating pre-generated licenses, see `com.adobe.flashaccess.samples.licensegen.GenerateLicense` in the Reference Implementation Command Line Tools "samples" directory.

Embedding licenses

Once content has been encrypted and a license has been pre-generated, the license may be embedded into the encrypted content.

To embed a license, obtain an instance of `com.adobe.flashaccess.sdk.media.drm.contentupdate.MediaKeyMetaDataUpdater`. If you know the type of the encrypted content, use the constructor for `FLVKeyMetaDataUpdater` or `F4VKeyMetaDataUpdater`; otherwise, use `MediaProcessorFactory.getMediaProcessor()` to return an instance based on the file type detected. Construct a `KeyMetaDataCallback` and invoke `modifyKeyMetaData()`. Your callback implementation will be invoked when the DRM metadata is located in the encrypted content. Based on the metadata found, you can choose a license to embed and set the license using `EmbedLicenseKeyMetaData.setEmbeddedLicenses()`.

For sample code demonstrating embedded licenses, see `com.adobe.flashaccess.samples.licenseembedder.EmbedLicense` in the Reference Implementation Command Line Tools "Samples" directory.

Note: *Adobe Access 2.0 clients will ignore any licenses embedded in the content and will attempt to obtain a license from the license server specified in the metadata. However, if the metadata indicates that no license server is available, an Adobe Access 2.0 client will need to upgrade to view the content.*

See "Out-of-band Licenses" on page 8.

Chapter 6: Implementing the License Server

In order to issue licenses to clients, you must deploy an Adobe Access license server. The license server uses the Adobe® Access™ SDK to perform these tasks:

- Process authentication requests, if username/password authentication is supported.
- Process license requests
- Process Get Server Version requests -- All servers must implement support for this type of request.
- Process domain registration requests -- Only needed if implementing a domain server.
- Process domain de-registration requests -- Only needed if implementing a domain server.
- Process synchronization -- Only needed if licenses specify Sync requirements.

In addition, the server needs to provide business logic for authenticating users, determining if users are authorized to view content, and optionally track license usage.

For details about the Java API discussed in this chapter, see *Adobe Access API Reference*.

License Server deployment options

The License Server can be deployed using one of the following options:

- **Adobe Access Server for Protected Streaming** -- This License Server is optimized for streaming. For instance, you can set up the server for Adobe HTTP Dynamic Streaming with Adobe Access. This server can be deployed easily with very little configuration required and will support multiple tenants, and can achieve a high level of scalability. However, since this implementation is optimized for streaming, it does not support the full Adobe Access features. For instance, username/password authentication, domains, and license chaining are not supported. The usage rules in licenses issued by this server are controlled through a server configuration file, which overrides the policy used at packaging time. See the *Adobe Access Server for Protected Streaming Guide* for more details on the usage rules supported by this server.
- **Reference Implementation License Server** -- Use this setup as a starting point for a custom server implementation. This is an example license server implementation, including source code, which demonstrates how to use the APIs in the Adobe Access SDK to handle all types of requests and how to implement custom business logic backed by a database. The usage rules in licenses issued by this server are controlled through the policy associated with the content at packaging time.
- **Custom Server Implementation** -- You can also implement your own licensing server using the SDK. The information in this chapter describes the APIs used to implement a license server.

Processing Adobe Access requests

The general approach to handling requests is to create a handler, parse the request, set the response data or error code, and close the handler.

The base class used to handle single request/response interaction is `com.adobe.flashaccess.sdk.protocol.MessageHandlerBase`. An instance of the `HandlerConfiguration` class is used to initialize the handler. `HandlerConfiguration` stores server configuration information, including transport credentials, timestamp tolerance, policy update lists, and revocation lists.The handler reads the request data and parses the request into an instance of `RequestMessageBase`. The caller can examine the information in the request and decide whether to return an error or a successful response (subclasses of `RequestMessageBase` provide a method for setting response data).

If the request is successful, set the response data; otherwise invoke `RequestMessageBase.setErrorData()` on failure. Always end the implementation by invoking the `close()` method (it is recommended that `close()` be called in the `finally` block of a `try` statement). See the `MessageHandlerBase` API reference documentation for an example of how to invoke the handler.

Note: *HTTP status code 200 (OK) should be sent in response to all requests processed by the handler. If the handler could not be created due to a server error, the server may respond with another status code, such as 500 (Internal Server Error).*

The client uses the License Server URL specified at packaging time as the base URL for all requests sent to the license server. For example, if the server URL is specified as "http://licenseserver.com/path", the client will send requests to "http://licenseserver.com/path/flashaccess/...". See the following sections for details on the specific path used for each type of request. When implementing your license server, be sure the server responds to the paths required for each type of request.

A license request can contain an authentication token. If username/password authentication was used, the request may contain an `AuthenticationToken` generated by the `AuthenticationHandler`, and the SDK will ensure the token is valid before issuing a license.

Using machine identifiers

All Adobe Access requests (with the exception of requests supporting FMRMS compatibility) contain information about the machine token issued to the client during individualization. The machine token contains a Machine Id, an identifier assigned during individualization. Use this identifier to count the number of machines from which a user has requested a license or joined a domain.

There are two ways of using the identifier. The `getUniqueId()` method returns a string assigned to the device during individualization. You can store the strings in a database and search by identifier. However, this identifier will change if the user reformats the hard drive and individualizes again. This identifier will also have a different value between Adobe® AIR® and Adobe® Flash® Player in different browsers on the same machine.

To more accurately count machines, you can use `getBytes()` to store the whole identifier. To determine if the machine has been seen before, get all the identifiers for a user name and call `matches()` to check if any match. Because the `matches()` method must be used to compare the values returned by `MachineId.getBytes`, this option is only practical when there are a small number of values to compare (for example, the machines associated with a particular user).

User authentication

An Adobe Access request can contain an authentication token.

If username/password authentication was used, the request may contain an `AuthenticationToken` generated by the `AuthenticationHandler`. To access and verify the token, use `RequestMessageBase.getAuthenticationToken()`. To initiate a username/password request on the client, use the `DRMManager.authenticate()` ActionScript or iOS API.

If the client and server are using a custom authentication mechanism, the client obtains an authentication token through some other channel and sets the custom authentication token using the `DRMManager.setAuthenticationToken` ActionScript 3.0 API. Use `RequestMessageBase.getRawAuthenticationToken()` to get the custom authentication token. The server implementation is responsible for determining whether the custom authentication token is valid.

Replay protection

For replay protection, it may be prudent to check whether the message identifier has been seen recently by calling `RequestMessageBase.getMessageId()`. If so, an attacker may be trying to replay the request, which should be denied. To detect replay attempts, the server can store a list of recently seen message ids and check each incoming request against the cached list. To limit the amount of time the message identifiers need to be stored, call `HandlerConfiguration.setTimestampTolerance()`. If this property is set, the SDK will deny any request that carries a timestamp more than the specified number of seconds off the server time.

Rollback detection

For rollback detection, some usage rules require the client to maintain state information for enforcement of the rights. For example, to enforce the playback window usage rule, the client stores the date and time when the user first began viewing the content. This event triggers the start of the playback window. To securely enforce the playback window, the server needs to ensure that the user is not backing up and restoring the client state in order to remove the playback window start time stored on the client. The server does this by tracking the value of the client's rollback counter. For each request, the server gets the value of the counter by calling `RequestMessageBase.getClientState()` to obtain the `ClientState` object, then calling `ClientState.getCounter()` to obtain the current value of the client state counter. The server should store this value for each client (use `MachineId.getUniqueId()` to identify the client associated with the rollback counter value), and then call `ClientState.incrementCounter()` to increase the counter value by one. If the server detects that the counter value is less than the last value seen by the server, the client state may have been rolled back. For more information on client state tamper detection, see the `ClientState` API reference documentation.

Global server configuration data

In addition to configuration used by the license server, `HandlerConfiguration` stores configuration information that can be sent to the client to control how licenses are enforced. This is done by creating a `ServerConfigData` class and calling `HandlerConfiguration.setServerConfigData()` (these settings apply only to licenses issued by this license server). The clock windback tolerance is one property that can be set by the license server to control how the client enforces licenses. By default, users may set their machine clock back 4 hours without invalidating licenses. If a license server operator wishes to use a different setting, the new value can be set in the `ServerConfigData` class. When you change the value of any of these settings, be sure to increment the version number by calling `setVersion()`. The new values will only be sent to the client if the version on the client is less than the current `ServerConfigData` version.

Crossdomain policy file

If the license server is hosted on a different domain than the video playback SWF, then a cross-domain policy file (crossdomain.xml) is necessary to allow the SWF to request licenses from the license server. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from other domains. Any SWF file served from a domain specified in the license server's cross-domain policy file is permitted to access data or assets from that license server.

Adobe recommends that developers follow best practices when deploying the cross-domain policy file by only allowing trusted domains to access the license server and limiting the access to the license sub-directory on the web server.

For more information on cross-domain policy files, please see the following locations:

- Web site controls (policy files)
- Cross-domain policy file specification

Handling Get Server Version requests

Adobe Access client 3.0 and higher send a Get Server Version request in order to determine the capabilities of the server. All servers using Adobe Access SDK 3.0 and higher must implement support for Get Server Version requests.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.getversion.GetServerVersionHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.getversion.GetServerVersionRequestMessage`
- The request URL must be “License Server URL in metadata” + “/flashaccess/getServerVersion/v3”

For Adobe Access SDK 4.0, the response to a Get Server Version request indicates to clients that the server supports versions 3 and 4 of the Adobe Access protocol.

While no custom logic is required to process Get Server Version requests, all the features described in “Processing Adobe Access requests” on page 23 are available for these requests.

Handling Domain Registration requests

If the DRM metadata indicates that domain registration is needed to play the content, the client application should invoke the DRMManager.addToDeviceGroup() ActionScript API or joinLicenseDomain() iOS API. If the client has not yet registered with the specified domain server (or if the application is forcing a re-join), a domain registration request is sent. The domain server determines whether the client is permitted to join a domain and issues one or more domain credentials to the client.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.domain.DomainRegistrationHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.domain.DomainRegistrationRequestMessage`
- If both the client and server support protocol version 4, the request URL is "Domain Server URL in metadata: + "/flashaccess/domain/v4". Otherwise, the request URL is Domain Server URL in metadata” + “/flashaccess/domain/v3”

When initializing the `DomainRegistrationHandler`, the domain server URL must be specified. This URL will be included in the domain tokens issued by the handler to indicate the domain server that issued the token. The URL must match the domain server URL specified in any policy that requires domain registration with the server.

To determine whether the client is permitted to join the domain, the server may examine the machine and user information in the request. See "Using machine identifiers" on page 24 for information on how to identify and count machines joining the domain. The server may also determine which domain the client is permitted to join. Note that a client may only be a member of one domain per domain server URL. If the machine already has a domain token for this domain server URL, the domain registration request will include the current domain name (`getRequestDomainName()`). For a re-join request, the domain server must either return the current set of domain credentials for this domain or return an error (the domain server may not return the domain credentials for a different domain).

If the domain server requires authentication to join a domain, the request should contain an authentication token. Just as with a license request, domain registration may require username/password or custom authentication. See "User authentication" on page 24 for details on handling authentication tokens.

The domain server is responsible for storing and managing the domain keys associated with each domain. When a new key pair needs to be generated for a domain (the first domain registration for the domain), invoke `generateDomainCredential(String, int, Principal, Date)`. This method will generate a new domain key pair and domain certificate. The domain server must store the private key and certificate and provide those objects when processing subsequent requests for this domain. It is also possible to generate a new key pair for a domain in order to roll over the keys. When rolling over the keys for a particular domain, be sure to increment the key version in `generateDomainCredential` as well.

Each subsequent time a machine registers with the same domain, the same domain private key and certificate should be used. Invoke `addDomainCredential(DomainToken, PrivateKey)` to return an existing domain credential to the client. If there are multiple domain credentials for the domain because the domain keys were changed, the server should provide domain credentials for the current domain key and all previous domain keys so the client can consume licenses bound to older domain keys. This enables a license bound to an old domain token to be moved to a newly registered machine and still be playable.

Handling Domain De-Registration requests

If the client application needs to leave a domain, it invokes the `DRMManager.removeFromDeviceGroup()` ActionScript API or the `leaveLicenseDomain()` iOS API to initiate the domain de-registration process. Domain de-registration is a two phase process. The client first sends a de-registration preview request. The domain server examines the request and determines whether the client is permitted to leave the domain (an error may be returned if the machine is not currently part of the domain). Upon a successful response, the client deletes its domain credentials and any licenses issued to that domain, then sends a de-registration request.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.domain.DomainDeRegistrationHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.domain.DomainDeRegistrationRequestMessage`
- If both the client and server support protocol version 4, the request URL is "Domain Server URL in metadata: + "/flashaccess/dereg/v4". Otherwise, the request URL is Domain Server URL in metadata" + "/flashaccess/dereg/v3"

When processing domain de-registration requests, the server uses `getRequestPhase()` to determine whether the client is in the preview or de-registration phase. In the preview phase, the domain server examines the request and determines whether the client is permitted to leave the domain (the request may be denied, for example, if the machine is not currently part of the domain). In the de-registration phase, the server records the fact that the machine has left the domain. The server is highly recommended to evaluate the same logic in the preview request as in the actual de-registration request to minimize the chance of the second phase failing.

Handling authentication requests

The `AuthenticationHandler` class is used to process authentication requests. It is used only for username/password authentication.

When generating the authentication token, the token expiration date must be specified. Custom properties may also be included in the token. If set, those properties will be visible to the server when the authentication token is sent in subsequent requests. See "Handling license requests" on page 28 for information on handling custom authentication tokens.

The handler reads an authentication request and parses the request message when `parseRequest()` is called. The server implementation examines the user credentials in the request, and if the credentials are valid, generates an `AuthenticationToken` object by calling `getRequest().generateAuthToken()`. If `AuthenticationRequestMessage.generateAuthToken()` is not called before `close()`, an authentication failure error code is sent.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.authentication.AuthenticationHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.authentication.AuthenticationRequestMessage`
- If both the client and server support protocol version 4, the request URL is "License Server URL in metadata: + "/flashaccess/authn/v4". If protocol version 3 is the maximum supported by either the client or server, Adobe Access clients will send authentication requests to "License Server URL in metadata" + "/flashaccess/authn/v3". Otherwise, authentication requests are sent to "License Server URL in metadata" + "/flashaccess/authn/v1"

Handling license requests

To request a license, the client sends the metadata that was embedded in the content during packaging. The license server uses the information in the content metadata to generate a license.

The `LicenseHandler` reads a license request and parses the request. `LicenseHandler` extends `BatchHandlerBase` to accommodate batch license requests, although this feature is not currently supported by Adobe Access clients. The `getRequests()` method will return a List of `LicenseRequestMessage` objects. The caller should iterate through the `LicenseRequestMessages`, and for each request, either generate a license or set an error code (see the `LicenseRequestMessage` API reference documentation for details). For each license request, the server determines whether it will issue a license. Call `LicenseRequestMessage.getContentInfo()` to obtain information extracted from the content metadata, including the content ID, license ID, and policies.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.license.LicenseHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.license.LicenseRequestMessage`
- If both the client and server support protocol version 4, the request URL is "License Server URL in metadata: + "/flashaccess/license/v4". If protocol version 3 is the maximum supported by either the client or server, Adobe Access clients will send authentication requests to "License Server URL in metadata" + "/flashaccess/license/v3". Otherwise, authentication requests are send to "License Server URL in metadata" + "/flashaccess/license/v1"

If an error occurs while parsing the request, a `HandlerParsingException` is thrown. This exception contains error information to be returned to the client. To retrieve the error information, call `HandlerParsingException.getErrorData()`. If an error occurs while generating a license because the policy requirements have not been satisfied, a `PolicyEvaluationException` is thrown. This exception also includes `ErrorData` to be returned to the client. See the API documentation for `LicenseRequestMessage.generateLicense()` for details on how policies are evaluated during license generation.

Licenses and errors are sent at one time when `LicenseHandler.close()` is called.

A device may have multiple licenses for the same content (same License ID), but can only have one license for a particular License ID and Policy ID. If it receives a license with a duplicate LicenseID/PolicyID, the new license will replace the old one only if the new license's issue date is later than the existing license's issue date. This logic is used to process licenses embedded into content, therefore, it is not recommended to embed more than one license with the same Policy ID in a chunk of content. The same logic applies to licenses passed to the client through the `DRMManager.storeVoucher()` ActionScript3 API; if the client already possesses a license with a later issue date, the provided license may be ignored.

Generating licenses

To issue a leaf license to the user, the SDK must decrypt the CEK contained in the content metadata and re-encrypt it for the machine requesting a license. To decrypt the CEK, the server must provide information required to decrypt the key. Call `ContentInfo.setKeyRetrievalInfo()` and provide an `AsymmetricKeyRetrieval` object. If the metadata contains multiple policies, the server must determine which policy to use and call `LicenseRequestMessage.setSelectedPolicy()`. Then call `LicenseRequestMessage.generateLicense()` to generate the license. Using the `License` object that is returned, you may modify the expiration or rights in the license.

License chaining

If the policy used to generate the license supports license chaining, the server must decide whether to issue a Leaf license, Root license, or both. To determine what type of license chaining a policy supports, use `Policy.getLicenseChainType()`, or call `Policy.getRootLicenseId()` to determine if the policy has a root license. With Adobe Access 2.0 license chaining, the server typically issues a leaf license the first time the user requests a license for a particular machine and a root license thereafter. To determine if the machine already has a leaf license for the specified policy, call `LicenseRequestMessage.clientHasLeafForPolicy()`.

Enhanced License Chaining

With enhanced license chaining in Adobe Access 3.0, it is recommended to issue both a Leaf and a Root the first time the user requests a license for a particular machine. If the user already has the Root license, the server may issue only a Leaf (call `LicenseRequestMessage.clientHasEnhancedRootForPolicy()` to determine if the client already has a 3.0 Enhanced Root). For subsequent license requests, the client will indicate that it already has a Leaf and a Root, so the server should issue a new Root license. When the enhanced license chaining is used, `setRootKeyRetrievalInfo()` must be called to provide the credentials needed to decrypt the root encryption key in the policy.

Note: *If the policy supports 3.0 Enhanced License Chaining, but the client is Adobe Access 2.0, the server will issue a 2.0 original chained license. To determine the client version, use LicenseRequestMessage.getClientVersion().*

Issuing Domain-bound licenses

In order to issue a license using a policy that requires domain registration, the client's request must include a valid domain token issued by the domain server specified in the policy. When the client requests a license, it will automatically include its domain tokens for any domain server specified in the content metadata, if it has registered with those domain servers. If the selected policy requires domain registration, the SDK will automatically select a domain token from the request to bind the license to, or return an error if no appropriate domain token was found.

A domain token is considered to be valid if it is not expired and if it was issued by an authorized Domain CA. The license server must specify the domain authorities from which it will accept domain tokens by configuring `HandlerConfiguration.setDomainCAs()`. If no Domain CAs are configured, the license server will not be able to issue domain-bound licenses.

If the metadata includes multiple policies, the license server business logic could select a policy based on whether the client presented a domain token. Use `LicenseRequestMessage.getDomainTokens()` to determine the domains with which the client has registered.

Issuing licenses for remote key delivery to iOS clients (requires Adobe Primetime)

New in 4.0

In order to issue licenses for content that requires Remote Key delivery for iOS devices, the Key Server's License Server certificate must be specified in `HandlerConfiguration.setKeyServerCertificate()`.

Minimum Client Version

Adobe Access 2.0.2 and higher introduce some new usage rules, which are not understood by Adobe Access 2.0 clients. By setting the minimum supported client version (`HandlerConfiguration.setMinSupportedClientVersion()`), the license server can control how older clients will behave when they encounter licenses with these usage rules. Based on this setting, the server can indicate whether older clients can ignore the usage rules they do not understand or whether older clients will not be able to consume licenses with those usage rules.

For example,

- If the license specifies the device capabilities requirements ("Device capabilities required to play protected content" on page 4), Adobe Access clients 2.0.2 and higher can enforce those requirements.
- If the license server does not want content to play on clients that do not understand the device capabilities requirements , set the minimum supported client version to 2 (for 2.0.2). This will prevent the server from issuing licenses to Adobe Access clients before 2.0.2. The minimum client version will also be enforced if the license is transferred from one client to another.
- If the license server wants to allow older clients to ignore the device capabilities requirements, set the minimum supported client version to 1 (for Adobe Access 2.0). The server will issue a license to any client version 2.0 and higher. If the client upgrades or transfers the license to another client with version 2.0.2 or higher, the device capabilities requirements in the license will be enforced, since the client would now support that usage rule.

License preview

The client can send a license preview request, meaning that the application can carry out a preview operation before asking the user to buy the content in order to determine whether the user's machine actually meets all the criteria required for playback. *License preview* refers to the client's ability to preview the license (to see what rights the license allows) as opposed to previewing the content (viewing a small portion of the content before deciding to buy). Some of the parameters that are unique to each machine are: outputs available and their protection status, the runtime/DRM version available, and the DRM client security level. The license preview mode allows the runtime/DRM client to test the license server business logic and provide information back to the user so he can make an informed decision. Thus the client can see what a valid license looks like but would not actually receive the key to decrypt the content. Support for license preview is optional, and only necessary if you implement a custom client that uses this functionality.

To determine whether the client sent a preview request or license acquisition request, call `LicenseRequestMessage.getRequestPhase()` and compare it to `LicenseRequestMessage.RequestPhase.Acquire`

Identity-bases licenses

If identity-based licensing is used, the server checks for a valid authentication token before issuing a license. See "User authentication" on page 24 for details on handling authentication tokens.

Note: *To preview a license for identity-based content, a client must authenticate.*

Updating policies

If policies are updated after the content is packaged, provide the updated policies to the license server so the updated version can be used when issuing a license. If your license server has access to a database for storing policies, you can retrieve the updated policy from the database and call `LicenseRequestMessage.setSelectedPolicy()` to provide the new version of the policy.

For license servers that do not rely on a central database, the SDK provides support for Policy Update Lists. A policy update list is a file containing a list of updated or revoked policies. When a policy is updated, generate a new Policy Update List and periodically push the list out to all the license servers. Pass the list to the SDK by setting `HandlerConfiguration.setPolicyUpdateList()`. If an update list is provided, the SDK consults this list when parsing the content metadata. `ContentInfo.getUpdatedPolicies()` contains the updated versions of policies specified in the metadata.

See Creating and updating policies and "Policy update lists" on page 17

Handling synchronization requests

New in 3.0. If a license specifies synchronization requirements ("Requirements for Synchronization" on page 2) the client will periodically send synchronization requests to the server, based on the frequency specified in the license. To enable synchronization messages, set SyncFrequencyRequirements in a PlayRight.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.sync.SynchronizationHandler`
- The request message class is `com.adobe.flashaccess.sdk.protocol.sync.SynchronizationRequestMessage`
- If both the client and server support protocol version 4, the request URL is "License Server URL in metadata: + "/flashaccess/sync/v4". Otherwise, the request URL is "License Server URL in metadata" + "/flashaccess/sync/v3"

Synchronization messages are used to synchronize the client's time with the server's time. If licenses are embedded in the content and do not need to be retrieved from a license server, synchronizing the client's time is important to prevent the client from modifying its clock in order to bypass the license expiration.

Synchronization messages can also be used to communicate the client state information to the server (`getClientState()`) for rollback detection. See "Processing Adobe Access requests" on page 23.

Handling FMRMS compatibility

There are two types of requests related to Flash Media Rights Management Server 1.x compatibility. One type of request is used to prompt 1.x clients to upgrade to a runtime that supports Adobe Access 2.0 or higher. Another is used to update 1.x metadata to the Adobe Access format before a license can be requested. Support for these requests is only needed if you previously deployed content using FMRMS 1.0 or 1.5.

Upgrading clients

If an FMRMS 1.x client contacts a Adobe Access server, the server needs to prompt the client to upgrade.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.compatibility.FMRMSv1RequestHandler`.
- The request URL is "*Base URL from 1.x content*" + "/edcws/services/urn:EDCLicenseService"

  Unlike other Adobe Access request handlers, this handler does not provide access to any request information or require any response data to be set. Create an instance of the `FMRMSv1RequestHandler`, and then call `close()`

Upgrading metadata

If an Adobe Access client encounters content packaged with Flash Media Rights Management Server 1.x, it will extract the encryption metadata from the content and send it to the server. The server will convert the FMRMS 1.x metadata into the Adobe Access format and send it back to the client. The client then sends the updated metadata in a standard Adobe Access license request.

- The request handler class is `com.adobe.flashaccess.sdk.protocol.compatibility.FMRMSv1MetadataHandler`.
- The request URL is "*Base URL from 1.x content*" +"/flashaccess/headerconversion/v1".

The metadata conversion could be done on the fly when the server receives the old metadata from the client. Alternatively, the server could preprocess the old content and store the converted metadata; in this case, when the client requests new metadata, the server just needs to fetch the new metadata matching the license identifier of the old metadata.

To convert metadata, the server must perform the following steps:

- Get `LiveCycleKeyMetaData`. To pre-convert the metadata, `LiveCycleKeyMetaData` can be obtained from a 1.x packaged file using `MediaEncrypter.examineEncryptedContent()`. The metadata is also included in the metadata conversion request (`FMRMSv1MetadataHandler.getOriginalMetadata()`).
- Get the license identifier from the old metadata, and find the encryption key and policies (this information was originally in the Adobe LiveCycle ES database. The LiveCycle ES policies must be converted to Adobe Access 2.0 policies.) The Reference Implementation includes scripts and sample code for converting the policies and exporting license information from LiveCycle ES.
- Fill in the `V2KeyParameters` object (which you retrieve by calling `MediaEncrypter.getKeyParameters()`).

- Load the `SigningCredential`, which is the packager credential issued by Adobe used to sign encryption metadata. Get the `SignatureParameters` object by calling `MediaEncrypter.getSignatureParameters()` and fill in the signing credential.
- Call `MetaDataConverter.convertMetadata()` to obtain the `V2ContentMetaData`.
- Call `V2ContentMetaData.getBytes()` and store for future use, or call `FMRMSv1MetadataHandler.setUpdatedMetadata()`.

Handling certificate updates

There might be times when you have to get a new certificate from Adobe. For example, when a production certificate expires, an evaluation certificate expires, or when you switch from an evaluation to a production certificate. When a certificate expires and you do not want to repackage the content that used the old certificate. You can make the License Server aware of both the old and new certificates.

Use the following procedure to update your server with the new certificates:

1 (Optional) When adding new entries to an existing policy update list or revocation list, be sure to sign with the new credentials, and use the old certificate to validate the signature on the existing file.

For example, use the following command line to add an entry to an existing policy update list, which was signed using a different credential:

```
java -jar AdobePolicyUpdateListManager.jar newList -f oldList oldSigningCert.cer -u pol 0 "" ""
```

2 Use the Java API to update the license server with the new policy update list or revocation list:

```
HandlerConfiguration.setRevocationList()
```

or:

```
HandlerConfiguration.setPolicyUpdateList()
```

In the reference implementation, the properties you use are `HandlerConfiguration.RevocationList` and `HandlerConfiguration.PolicyUpdateList`. Also update the certificate used to verify the signatures: `RevocationList.verifySignature.X509Certificate`.

3 To consume content that was packaged using the old certificates, the license server requires the old and new license server credentials and transport credentials. Update the license server with the new and old certificates.

For the license server credentials:

- Ensure that the current credential is passed into the `LicenseHandler` constructor:
  - In the reference implementation, set it through the `LicenseHandler.ServerCredential` property.
  - In the Adobe Access Server for Protected Streaming, the current credential must be the first credential specified in the `LicenseServerCredential` element in the flashaccess-tenant.xml file.
- Ensure that the current and old credentials are provided to `AsymmetricKeyRetrieval`
  - In the reference implementation, set it through the `LicenseHandler.ServerCredential` and `AsymmetricKeyRetrieval.ServerCredential.`*`n`* properties.
  - In the Adobe Access Server for Protected Streaming, the old credentials are specified after the first credential in the `LicenseServerCredential` element in the flashaccess-tenant.xml file.

For the transport credentials:

- Ensure that the current credential is passed into the `HandlerConfiguration.setServerTransportCredential()` method:
  - In the reference implementation, set it through the `HandlerConfiguration.ServerTransportCredential` property.
  - In the Adobe Access Server for protected streaming, the current credential must be the first credential specified in the `TransportCredential` element in the flashaccess-tenant.xml file.
- Ensure that the old credentials are provided to `HandlerConfiguration.setAdditionalServerTransportCredentials`():
  - In the reference implementation, set it through the `HandlerConfiguration.AdditionalServerTransportCredential.`*n* properties.
  - In the Adobe Access Server for protected streaming, this is specified after the first credential in the `TransportCredential` element in the flashaccess-tenant.xml file.

4 Update packaging tools to make sure they are packaging content with the current credentials. Ensure that the latest license server certificate, transport certificate, and packager credential are used for packaging.

5 To update the Key Server's License Server Certificate:

- Update the credentials in the Adobe Access Key Server tenant configuration file. Include both the old and new Key Server credentials in flashaccess-keyserver-tenant.xml.
- Ensure the current certificate is passed into the `HandlerConfiguration.setKeyServerCertificate()` method.
  - In the reference implementation, set it through the `HandlerConfiguration.KeyServerCertificate` property.
  - In the Adobe Access Server for Protected Streaming, specify the Key Server's certificate in the through the Configuration/Tenant/Certificates/KeyServer element.

Performance tuning

Use the following tips to help to increase performance:

- Using a network HSM can be significantly slower than using a directly-connected HSM.
- For improved performance, you can optionally enable native support for cryptographic operations by deploying the platform-specific libraries located in the "thirdparty/cryptoj" folder of the SDK. To enable native support, add the library for your platform (jsafe.dll for Windows or libjsafe.so for Linux) to the path.
- A 64-bit operating system, such as the 64-bit version of Red Hat® or Windows, provides much better performance over a 32-bit operating system.

Generating random numbers

Hardware random number generators can be used on Linux servers to ensure that sufficient entropy is generated. If the machine cannot generate enough entropy, Adobe Access operations that require a source of randomness will block while waiting for data from `/dev/random`.

Chapter 7: Revoking client credentials

Under certain conditions it is necessary to revoke a client's credentials or check whether a given set of credentials have already been revoked. Credentials may be revoked if the credentials are compromised. When this happens, licenses will no longer be issued to compromised clients.

Adobe maintains Certificate Revocation Lists (CRLs) for revoking compromised clients. These CRLs are automatically enforced by the SDK. License servers may further restrict clients by disallowing particular machine credentials or particular versions of DRM and runtime credentials. A `RevocationList` may be created and passed into the SDK to revoke machine credentials. Particular DRM/runtime versions can be revoked either at the policy level (by setting module restrictions in the play right) or globally (by setting module restrictions in the `HandlerConfiguration`).

The discussion in this chapter will be centered on revoking client credentials.

Certificate Revocation Lists published by Adobe

The SDK automatically enforces the CRLs published by Adobe. The SDK fetches the CRLs from Adobe and caches them locally. Several system properties are available to control how the CRLs are cached. To set these properties, use the `-D` option when starting Java to specify the property name and value.

By default, the CRLs will be saved to disk in the directory specified by "java.io.tmpdir" (if the temp dir property is not set, the working directory is used). The server must have read/write access to this directory. To use a directory other than the temp directory, specify a system property called "flashaccess.crl.dir" whose value is the directory where the CRLs should be stored.

By default, if a CRL cannot be retrieved, this is treated as an error. To ignore missing CRLs in a development environment, set the Java System property "flashaccess.crl.error=ignore". This property must not be set for production environments.

If an error occurs fetching the CRL, by default the SDK will not try to fetch the CRL again for 5 minutes. The duration can be configured by setting the number of minutes in the "flashaccess.crl.retry" System property. If the value is less than 1, the server will not wait before retrying (not recommended).

If a cached CRL is expiring soon, the SDK will attempt to fetch it again from the CRL server. By default, the SDK will try to fetch the CRL 15 days before its expiration. This duration is configurable by setting the number of minutes in the "flashaccess.crl.prefetch" system property. If the value is less than 1, the server will not try to pre-fetch the CRL (not recommended for production environments).

Revoking DRM client and runtime credentials

DRM/Runtime versions are identified by security level, version number, and other attributes including OS and runtime. To restrict the DRM/Runtime versions allowed, set the module restrictions in a policy or in a `HandlerConfiguration`. Module restrictions may include a minimum security level and list of module versions that are not permitted to be issued a license. See “Black-list of DRM Clients restricted from accessing protected content” on page 3 for details on the attributes used to identify a DRM/Runtime module.

If the minimum security level is set, the version on the client (specified in the machine token), must be greater than or equal to the specified value.

If a list of excluded versions is specified and the client's version matches any of the version identifiers in the list, the client will not be allowed to use a right containing this ModuleRequirements instance. For a module to match the version information, all parameters specified in the version information, except for the release version, must exactly match the modules's values. The release version matches if the client module's value is less than or equal to the value in the version information.

In the event a breach is reported with a particular DRM client or runtime version, the content owner and content distributer (who runs the license server) can configure the server to refuse to issue licenses during a period in which Adobe does not have a fix available. This can be configured through the `HandlerConfiguration` as described above, or by making changes to all the policies. In the latter case, you can maintain a policy update list and use it to check whether a policy has been updated or revoked.

If you require a newer version of the Adobe® Flash® Player/Adobe® AIR® Runtime or the Adobe Content Protection library (DRM module), update your policies as shown in "Updating a policy using the Java API" on page 15 and create a Policy Update List, or set restrictions in `HandlerConfiguration` by invoking `HandlerConfiguration.setRuntimeModuleRequirements()` or `HandlerConfiguration.setDRMModuleRequirements()`. When a user requests a new license with these blacklists enabled, the newer runtimes and libraries must be installed before a license can be issued. For an example on blacklisting DRM and runtime versions, see the sample code in "Updating a policy using the Java API" on page 15.

Revoking machine credentials

Adobe maintains a CRL for revoking machine credentials that are known to be compromised. This CRL is automatically enforced by the SDK. If there are additional machines to which you do not want your license server to issue licenses, you may create a machine revocation list and add the issuer name and serial number of the machine tokens you want to exclude (use `MachineToken.getMachineTokenId()` to retrieve the issuer name and serial number of the machine certificate).

Revoking machine credentials involves the usage of a `RevocationListFactory` object. To create a revocation list, load an existing revocation list, and check whether a machine token has been revoked by using the Java API, perform the following steps:

1. Set up your development environment and include all of the JAR files mentioned in "Setting up the development environment" on page 11 within your project.
2. Create a `ServerCredentialFactory` instance to load the credentials needed for signing. The license server credential is used to sign the revocation list.
3. Create a `RevocationListFactory` instance.
4. Specify the issuer and serial number of the machine token to be revoked by using a `IssuerAndSerialNumber` object. All Adobe Access requests contain a machine token.
5. Create a `RevocationList` object using the `IssuerAndSerialNumber` object you just created, and add it to the revocation list by passing it into `RevocationListFactory.addRevocationEntry()`. Generate the new revocation list by calling `RevocationListFactory.generateRevocationList()`.
6. To save the revocation list, you can serialize it by calling `RevocationList.getBytes()`. To load the list, call `RevocationListFactory.loadRevocationList()` and pass in the serialized list.
7. Verify that the signature is valid and the list was signed by the correct license server by calling `RevocationList.verifySignature()`.

8 To check whether an entry was revoked, pass the `IssuerAndSerialNumber` object into `RevocationList.isRevoked()`. The revocation list may also be passed into `HandlerConfiguration` to have the SDK enforce the revocation list for all authentication and license requests.

To add additional entries to an existing `RevocationList`, load an existing revocation list. Create a new `RevocationListFactory` instance, and be sure to increment the CRL number. Call `RevocationListFactioryEntries.addRevocationEntries` to add all the entries from the old list to the new list. Call `RevocationListFactory.addRevocationEntry` to add any new revocation entries to the RevocationList.

For sample code demonstrating how to create a revocation list, load an existing revocation list, and check whether a machine token has been revoked, see `com.adobe.flashaccess.samples.revocation.CreateRevocationList` in the Reference Implementation Command Line Tools "samples" directory.

Chapter 8: Creating video players

In order to play back protected content, your application must use the ActionScript 3 DRM APIs or Adobe Primetime. Please refer to *Programming ActionScript 3* and the *ActionScript 3.0 Reference for the Adobe Flash Platform* for more information. For building iOS-based video players, refer to *Adobe Primetime PSDK documentation*.