PACKAGING AND SIGNING ADOBE EXTENSIONS

TECHNICAL NOTE

Adobe

Packaging and Signing Extensions

About Adobe® Application Extensions

The Adobe Extension SDK provides developers with a consistent platform in which to develop and deploy extensions across the suite. Adobe Application Extensions run in much the same way in different Adobe desktop applications, providing users with a rich and uniform experience.

Adobe Application Extensions use HTML and JavaScript to create cross-platform user interfaces. Extensions have access to the host application's scripting interface, and can use the host's scripting DOM to interact with the application. ExtendScript is Adobe's extended version of ECMA JavaScript. The host applications that have ExtendScript DOMs are packaged with the ExtendScript Toolkit, which allows you to develop and debug ExtendScript code.

Packaging and signing your extension for deployment

The Extension Manager allows users to install the extension you are developing on machines other than the one you are currently using (across platforms), to share the extension with other users, and to distribute it to customers.

Extenstion Manager requires that an extension be packaged in the ZXP format, and the package must be signed with a valid certificate and time-stamped.

IMPORTANT NOTE ABOUT TIME STAMPS: Digital certificates typically have a lifetime of one to four years, and long-lasting certificates can be prohibitively expensive. Without a time stamp, an extension that previously loaded and ran successfully can suddenly become unusable because a certificate that was valid when it was used to sign the extension has since expired.

You can add a time stamp using a free, self-signed certificate, and it has the effect of extending the validity of the signature. For this reason, we recommend that you always include a time stamp when signing your extension package. For details, see "How signing works" on page 5.

The package format

An Extension Manager package uses the ZXP format. This is an archive file with the extension `.zxp`, which contains:

- A copy of the `CSXS` folder containing the `manifest.xml` file.
- A copy of the folder containing the extension-panel HTML file and any dependant files.
- A copy of any other optional resources used by the extension, such as icons and localization files. For a hybrid extension, it must include the resource files for the native plug-in or scripting component.
- A file named `mimetype`, generated by the packaging and signing process.

Creating the deployment package

Adobe provides a number of packaging tools that help you to configure and create the ZXP package for your HTML/JS extension.

- Extension Builder 3 and Configurator 4 include easy-to-use packaging wizards for packaging extensions that you create with those tools.
- If you have a producer account with Adobe Exchange, the Adobe Exchange Packager is available from http://www.adobeexchange.com/resources.
- For all types of extensions and plug-ins, you can use the lower-level `ZXPSignCmd`, a command-line tool available from Adobe Labs.

Using the Extension Signing Toolkit

Adobe provides a command-line tool, `ZXPSignCmd`, that you can use to package and sign extensions so they can be installed in Adobe desktop applications using Extension Manager. See the Adobe Extension SDK page to download the toolkit for your platform.

After testing your extension thoroughly, you must package and sign your extension so users can install it in their systems using Extension Manager. To prepare for this step, it is recommended that you copy all of the files in the Output folder for your extension to a staging folder for ease of packaging. Make sure the staging folder contains a subfolder named `CSXS/`, which contains the `manifest.xml` file:

```
<staging_folder>/CSXS/manifest.xml
```

You can add any extra resources to the root or to a folder within the root folder. Within the manifest file, references to these resources should use pathnames that are relative to the root. For example, if your main panel HTML file is located at `<staging folder>/Simple.html`, the path in the manifest should be specified as `./Simple.html`.

Using ZXPSignCmd

You can use this tool to create a self-signed certificate, create a signed ZXP package, or verify an existing ZXP package.

- To create a signed package:

```
ZXPSignCmd -sign <inputDir> <outputZxp> <p12> <p12Password> [options]
```

inputDir		The path to the folder containing the source files to package.
outputZxp		The path and file name for the ZXP package.
p12		The signing certificate; see "How signing works" on page 5.
p12Password		The password for the certificate.
options	`-tsa <timestampURL>`	The timestamp server. For example: `https://timestamp.geotrust.com/tsa`

▶ To verify a ZXP package:

```
ZXPSignCmd -verify <zxp>|<extensionRootDir> [options]
```

zxp	The path and file name for the ZXP package.	
extensionRootDir	The path to the folder containing the deployed ZXP.	
options	`-certinfo`	If supplied, prints information about the certificate, including timestamp and revocation information.
	`-skipOnlineRevocationChecks`	If supplied, skips online checks for certificate revocation when `-certinfo` is set.
	`-addCerts <cert1> <cert2> ...`	If supplied, verifes the certificate chain and assesses whether the supplied DER-encoded certificates are included .

▶ To create a self-signed certificate:

```
ZXPSignCmd -selfSignedCert <countryCode> <stateOrProvince> <organization>
<commonName> <password> <outputPath.p12> [options]
```

countryCode *stateOrProvince* *organization* *commonName*	The certificate identifying information.	
password	The password for the new certificate.	
outputPath.p12	The path and file name for the new certificate.	
options	`-locality <code>`	If supplied, the locale code to associate with this certificate.
	`-orgUnit <name>`	If supplied, an organizational unit to associate with this certificate.
	`-email <addr>`	If supplied, an email address to associate with this certificate.
	`-validityDays <num>`	If supplied, a number of days from the current date-time that this certificate remains valid.

Example

If you already have a certificate, you can use that. Otherwise, begin by creating a self-signed certificate:

```
./ZXPSignCmd -selfSignedCert US NY MyCompany MyCommonName abc123 MyCert.p12
```

This generates a file named `MyCert.p12` in the current folder. You can use this certificate to sign your extension:

```
./ZXPSignCmd -sign myExtProject myExtension.zxp MyCert.p12 abc123
```

This generates the file `myExtension.zxp` in the current folder, adding these two files to the packaged and signed extension in the final ZXP archive:

- `mimetype` : A file with the ASCII name of `mimetype` that holds the MIME type for the ZIP container (`application/vnd.adobe.air-ucf-package+zip`).
- `signatures.xml`: A file in the `META-INF` directory at the root level of the container file system that holds digital signatures of the container and its contents.

How signing works

The signature verifies that the package has not been altered since its packaging. When the Extension Manager tries to install a package, it validates the package against the signature, and checks for a valid certificate. For some validation results, it prompts the user to decide whether to continue with the installation. In addition, CEP checks for a valid certificate each time a host application tries to run an extension.

Certificates used to cryptographically sign documents or software commonly have expiration duration between one and four years. If the certificate has no valid timestamp, and the certificate used to sign the extension has expired, the extension cannot be installed or loaded. There is no warning or notification to the user before the signature expires. To make your extension available to users again, you would have to repackage it with a new certificate.

A valid timestamp ensures that the certificate used to sign the extension was valid at the time of signing. For this reason, you should always add a timestamp to the signature when you package and sign your extension. A timestamp has the effect of extending the validity of the digital signature.

These are the possible validation results:

Signature	Signing certificate	Extension Manager action	CEP action
No signature	N/A	Shows error dialog and aborts installation	Extension does not run
Signature invalid	Any certificate	Shows error dialog and aborts installation	Extension does not run
Certificate used to sign has expired, and no timestamp	Any certificate	Shows error dialog and aborts installation	Extension does not run
Certificate used to sign has expired, but has a valid timestamp	Any certificate	Silently installs extension	Extension runs normally
Signature valid	Adobe certificate	Silently installs extension	Extension runs normally
	OS-trusted certificate	Silently installs extension	Extension runs normally
	other certificate	Prompts user for permission to continue the installation	Extension runs normally

To sign extensions, a code-signing certificate must satisfy these conditions:

- The root certificate of the code-signing certificate must be installed in the target operating system by default. This can vary with different variations of an operating system. For example, you may need to check that your root certificate is installed into all variations of Win XP, including home/professional, SP1, SP2, SP3, and so on.
- The issuing certificate authority (CA) of the code-signing certificate must permit you to use that certificate to sign extensions.

To make sure a code-signing certificate satisfies these conditions, check directly with the certificate authority that issues it.

The following CAs and code-signing certificates are recommended for signing extensions:

- GlobalSign
  - ObjectSign Code Signing Certificate
- Thawte
  - AIR Developer Certificate
  - Apple Developer Certificate
  - JavaSoft Developer Certificate
  - Microsoft Authenticode Certificate
- VeriSign
  - Adobe AIR Digital ID
  - Microsoft Authenticode Digital ID
  - Sun Java Signing Digital ID