Wraptool Manual

Overview 3

Wraptool 3

Wrapper Binaries 4

Wrap Configs 4

Experiences 4

Wrapper Data Management 4

Wrapper Periodic License Checks 5

Digital Signature Support 6

Apple Notarization Support 8

Wraptool License 8

Windows Wrapper Integration 8

Operations 10

Common Arguments 10

Argument Spaces 13

wraptool help or wraptool -h ***13***

wraptool clean ***13***

wraptool sync ***13***

wraptool list ***14***

wraptool wrap ***14***

wraptool info ***22***

wraptool simulate ***23***

wraptool sign ***24***

wraptool verify ***28***

wraptool dump ***29***

wraptool export ***29***

wraptool codegen ***30***

wraptool dcgen ***34***

wraptool encryptcontent ***41***

wraptool signcontent ***44***

wraptool repair ***46***

Overview

Wraptool

Wraptool is a command line executable that adds protection to your product (content or executable). This protection includes support for:

- licensing for controlling the use of your products,
- encryption for the privacy of your code and content, and
- digital signatures for tamper-proofing your products and for identifying your genuine products.

Wraptool is part of PACE's Eden protection system, that replaces InterLok and ilok.com. Because Wraptool is a command line executable, it can be easily integrated into software development systems. Unlike InterLok, the protection setup is done on the Eden Server called PACE Central. Wraptool communicates with the server to get the wrapper data that is needed to protect your product. This data can be cached and exported for offline use and for source control.

For protecting your executable, Wraptool can optionally add a wrapper (injected code) that can do the following before allowing your executable to run:

- check that a PACE license (auth, short for authorization) for your product is present and valid,
- check that your product has not be tampered with using both platform specific and PACE signatures,
- decrypt your product using keys and information in the auth for your product, and
- provide user interface assistance to guide end-users in obtaining an auth for your product.

By itself, the wrapper provides very good protection for your product. Combining the wrapper with PACE's Fusion protection, which integrates protection checks throughout your code, provides even better protection.

For your content protection, Wraptool can optionally encrypt and/or sign your content. A new PACE Eden interface, which is now a part of the fusion library, allows you to decrypt content, but only in the presence of a valid auth for your content. The interface also allows you to verify the signature of content and even executables. This interface also supports advanced licensing control, in case you need to do more than the wrapper provides. For an overview of the content protection, please see the "PaceEdenContentProtection.pdf" document, which is included in the PACE Eden SDK.

Wrapper Binaries

The wrapper code and the code that protects your executables and content are not actually part of Wraptool. Instead, this core code is in the wrapper binaries, which are provided to Wraptool by the PACE Central server. This means that updates to the protection process do not require running an installer. Instead, you can have multiple versions of wrapper binaries that you choose from. You can setup your protection options to use the latest wrapper binaries or can specify the exact wrapper binaries that you want to use. Furthermore, the selection of the binaries can be overridden by the WrapTool command line arguments.

Wrap Configs

Wrap Config is short for wrapper configuration. The Wrap Config holds all of the options that are applied to the wrapping process and the execution of the applied wrapper. Because a Wrap Config specifies the auths that will allow the use of your product, a Wrap Config can also be applied to content protection, although many of the wrapper specific options are ignored. Wrap Configs are created on the PACE Central Server and are downloaded by Wraptool. Then, Wraptool operations use your selected Wrap Config to protect your product with your chosen options.

Typically, you would create a Wrap Config for each product that you want to sell separately, and then when using Wraptool commands, you reference a product's Wrap Config when protecting the product executable or content. Then to use the protected product executable or content, the auth corresponding to an allowed product in the Wrap Config must be present.

A Wrap Config can involve multiple products. For example, if you have a Pro product and Lite product, you can create a Lite product Wrap Config that allows the Pro product auth and the Lite product auth. Protecting the Lite product executable or content with this Wrap Config would allow this executable or content to be used in the presence of either the Pro product auth or Lite product auth. Of course the, Pro product Wrap Config would only allow the Pro product auth, so that Pro product executable or content could only be used in the presence of the Pro product auth.

Experiences

The Experience is the graphics and text that an end-user will see (will experience) when the wrapper has to present user interface items. Experiences are created in the PACE Central Server and are downloaded by Wraptool. Then, Wraptool operations use your selected Experience in the wrapped executables that will show the wrapper's user interface.

Wrapper Data Management

Since wrapper data, like Wrapper Binaries, Wrap Configs and Experiences, are all created on the PACE Central Server, Wraptool operations have arguments for downloading this data from the server. Once downloaded, this data resides in your wrapper cache, a folder on your computer's hard disk. If you don't need any new updates from the server, you can use Wraptool completely offline by using the wrapper data in your cache. Wraptool can export data from the server or

from your wrapper cache to a location of your choice. This is handy for adding wrapper data to your source code control. Any exported wrapper data can be used by Wraptool just like the wrapper data from the server or the wrapper cache.

Wrapper Periodic License Checks

Starting with the v5 Eden SDK, we support periodic license checks for wrapped binaries. When a wrapped binary is successfully authorized, a license check will be performed periodically on behalf of the wrapped binary. This mechanism will optionally call a developer provided callback to inform the wrapped binary regarding license changes.

If you're using the wrapper without Fusion, we recommend that you implement this callback function. If your callback is told that a license was not found, you can either disable the product as appropriate if running in a shared environment, or quit if your product is a monolithic application. You should not depend on the periodic license check to find a license on behalf of end-users because license checks are infrequent (see below).

If you're using Fusion, then there is little need to provide this callback function because Fusion will automatically perform periodic license checks at runtime. That said, there's no harm with providing this callback and using it to call into one of your Fusion failure handlers. If you're using Fusion Hardware Breakpoint Defense, especially in non-shared mode, you may also need to annotate your callback with the PACE_FUSION_HWBP_GATEWAY macro.

License checks are performed infrequently on behalf of the wrapped product. We've chosen the frequency based on a number of factors, including the anticipated load on our iLok Cloud services and the current lease times for floating licenses provided by an iLok License Server on the LAN.

A license is first checked at 25 minutes from launch. If the license was found on any location other than a floating iLok License Server, subsequent checks are an hour apart; the second check is 1:25 hours from launch. In contrast, floating licenses found on an iLok License Server are checked about once every 25 minutes in order to maintain the floating license lease. If you start more than one client on the same machine using the same license, they each have a separate timer.

Please note that the periodic check feature is not implemented on Windows 7.

Since the wrapper only registers your product for periodic checks when a license was found, the initial state of your product will be authorized. Your callback will only be called if a periodic license check fails. Subsequently your callback may be called again if the license is provided by the end-user again.

IMPORTANT! Your callback function will be called from a thread other than the main thread. It is important that your implementation takes this into account and uses the appropriate thread synchronization mechanisms when accessing objects and resources shared across threads in your product.

Also, since the calling thread belongs to an Eden component and could be shared among other products in the process, your callback implementation should quickly return to the caller. Do not perform any blocking operations in the callback itself.

The periodic license check mechanism does not provide any visible user interface. The UI is left up to the developer. Since your callback will be called from a thread owned by an Eden component, you should defer the display of any blocking UI to another thread. If you have a main event loop, perhaps that would be the best place to tell the end-user that their license is missing. Optionally you can consider making PaceEden license check calls on your own to help the end-user get back to a licensed state.

Please see the “Wrapper Periodic License Checks” section of PaceEden.h for details on the callback function signature and parameters.

Digital Signature Support

Wraptool will optionally digitally sign your target binary after wrapping if the corresponding wrap config indicates that digital signing is required. You can also use wraptool to sign an executable without wrapping.

When digitally signing, wraptool uses a combination of both platform-specific and PACE proprietary digital signature technologies. The way this works is wraptool first signs using the appropriate platform signature technology, then wraptool signs that signature using credentials (from an iLok2 or iLok3) that have been issued to a PACE customer.

The idea is that first and foremost the resulting binary will appear to be well signed for the platform. That is, the operating system will think it's signed, and will not know about the PACE side. But PACE customers with access to the full Eden SDK can make simple, cross-platform Eden API calls to verify that the binary was signed correctly for the platform, and that it was signed by a PACE publisher. Additionally, you can optionally determine the publisher that performed the signing, the product that was signed, as well as other signing details.

On the Mac, a binary signed by wraptool uses the codesign tool and a signing identity in the keychain to perform the platform signature. Generally the signing identity should be an Apple-issued application developer ID for compatibility with Mountain Lion's Gatekeeper. For development systems with Xcode 4 and above, you will need to install the Xcode command line tools in order for codesign to be usable by wraptool.

On Windows, you should use an Authenticode certificate from a Microsoft approved certificate authority. See Microsoft's documentation for information regarding Authenticode and approved providers. In order to sign under Windows, you will either need to provide the certificate and private key in PKCS12 format (a standard format for password encrypting certified key pairs) or a SHA1 thumbprint (string of 40 characters) of the certificate if your Authenticode certificate resides in the Windows Certificate Manager. Generally PKCS12 files are encrypted with a password, in which case you

must also provide the same password to wraptool when wrapping or signing. A signing certificate has to be stored in "Personal" certificate store of the current user in order to be used by wraptool.

At the time of this writing, "Extended Validation" certificates require a hardware dongle from a manufacturer other than PACE. If you use an EV certificate, you must follow the certificate provider's instructions for installing the certificate in your personal certificate store. Once done, you can use the wraptool --signid option to reference the EV certificate by its SHA1 thumbprint. Under the hood wraptool will call signtool.exe, which in turn knows how to use the EV certificate dongle to provide the Windows level code signature. Please refer to the documentation from your EV certificate provider for details regarding installing and using your EV certificate and dongle.

The thumbprint of a signing certificate can be obtained through a couple of different methods. Here's how to use the Microsoft management console to obtain the thumbprint for a certificate installed in your personal certificate store:

1. Run Microsoft management console (mmc.exe), add certificate snap-in for "My user account".
2. In the certificate snap-in, open the "Personal" certificate store.
3. Double click the signing certificate in store. A window will appear on the screen that displays the information of the certificate.
4. On the "Detail" page, if the value of the field called "Thumbprint algorithm" is "sha1", the value of the field called "Thumbprint" is the data that can be converted to a certificate thumbprint used by wraptool, by simply removing the space characters in the string. For example, suppose "88 05 b3 59 ed ea 7e 86 02 6f 2c 7a 33 d9 ee 29 7d b1 91 de" is the value of the field called "Thumbprint", "8805b359edea7e86026f2c7a33d9ee297db191de" will be the certificate thumbprint used by wraptool.

Alternatively you can use certutil.exe directly on your certificate file:

1. Open a console window.
2. Execute this command line (substituting the path to your certificate file):

   certutil.exe –v –dump path_to_certificate_file

3. The output will show the detailed description of the certificate. A piece of the description called "Cert Hash (sha1)" can be converted to a certificate thumbprint used by wraptool, by simply removing space characters in the string.

Note that wraptool and the Eden digital signature APIs do not require that you use an Authenticode or Extended Validation certificate. Effectively you can use whatever certificate you like, including a self-signed certificate. We allow this because our digital signature implementation is tolerant of the errors from Microsoft's APIs that happen when a signed binary has no root certificate installed. However, since signing with an untrusted certificate may result a binary that the Microsoft OS might not trust, we strongly recommend you use a genuine Authenticode certificate.

You probably only need an EV certificate when signing an application that would be launched directly by end users. EV certificates avoid Windows 8 SmartScreen warnings. If you are signing a plug-in, or other add in to an existing application, then you probably do not need an EV certificate at this time.

Before deciding on what kind of certificate you need for Microsoft code signing, please familiarize yourself with the options by reading Microsoft's documentation. PACE can only make basic recommendations in this regard, and Microsoft may change their requirements at any time.

In addition to signing as part of the wrapping process, you can arbitrarily sign a binary using the "sign" operation. Use the "verify" operation to verify the signature of a binary that has been signed as part of the wrapping process, or explicitly signed with the "sign" operation.

Apple Notarization Support

Wraptool will optionally notarize your digitally signed target binaries. For details, please see the "macOS Notarization Support" section of the GettingStartedGuide.html. The options that support notarization are summarized in the "wraptool wrap" and "wraptool sign" sections of this document.

Wraptool License

To run wraptool, you will need an Eden Tools license on an iLok2, iLok3 or iLok Cloud. If you are running wraptool on multiple computers at the same time, then you will need a license for each computer. Since iLok Cloud for a given account can only be active on one computer at a time, you will need an account for each computer, and each account will need its own Eden Tools license. When you have your account(s) setup, you can contact support@paceap.com for additional Eden Tools licenses.

Windows Wrapper Integration

To provide the highest level of security, the wrapper needs to integrate with the actual code of your binary. On the mac, there are ways to achieve this automatically. On Windows, however, it is necessary to compile a simple support file into your binary. This file, named 'pacefusionsup.c', is located in the PACE SDK directory "$(PACE_FUSION_HOME)\Inject\win32". Please add this file to your project, and ensure that it is compiled as a "C" file by using the setting "Compile as C code" (/TC).

NOTE: Because Fusion protected programs already include 'pacefusionsup.c' in their projects, they automatically support the highest level of wrapper security.

If wraptool does not detect that 'pacefusionsup.c' has been compiled into your project, it will issue a warning that you are not getting the highest level of wrapper security:

```
Warning: the highest level of wrapper security cannot be used because 'pacefusionsup.c' has not been properly
compiled into the target binary. Please see 'Windows wrapper integration' in WraptoolManual.pdf for more details.
```

In this case, the wrapper will still provide a base level of security, but it will not be as secure.

If wraptool detects that ‘pacefusionsup.c’ has been compiled into your project, it will confirm this by issuing the following status message:

```
Note: wraptool has detected that 'pacefusionsup.c' was compiled into the target binary, so the highest level of
wrapper security is being used. Please see 'Windows wrapper integration' in WraptoolManual.pdf for more details.
<Diagnostic 1> ...
<Diagnostic 2> ...
```

The “<Diagnostic X>” messages are provided to assist PACE technical support in case there are wrapping issues.

Operations

Common Arguments

The command line format for executing a wraptool operation is as follows:

```
wraptool operationName argumentList
```

The operationName, which is required, includes: help, clean, sync, list, dump, export, wrap, info, simulate, sign, verify, codegen, dcgen, encryptcontent, signcontent and repair. These operations are described in the sections below.

The argumentList can be zero, one or more arguments. Here are the commonly used arguments that are used across many commands:

--verbose
: The verbose argument, which works with any operation, causes Wraptool to display extra information, if available, about the operation as the operation progresses.

--account MyUserAccountName
: This tells the PACE Central server the name of the user account that is requesting wrapper data. The user account has to have the privileges to download wrapper data for at least one publisher. Otherwise, the user will not be able to download any wrapper data.

--password MyPassword
: This tells the PACE Central server the password of the user account. Putting the password in a command line script would not be very secure. So, all you have to do is present your password once, and the password will be automatically captured in the keychain or secure storage of your computer, so that you don't need to provide this argument anymore, unless you change your password. Logging into your computer is what unlocks access to this password in the keychain or secure storage of your computer.

--force
: When providing an account and password for the purposes of downloading wrapper data, the --force argument causes data needed for the operation to be downloaded from the server replacing any corresponding data in your wrapper cache. If you don't use the --force argument, then the wrapper data should only be downloaded if the server indicates that new data is available for your operation. So really, the --force argument shouldn't be needed, but it is available just in case.

--allowsigningservice
For performance reasons, using iLoks for signing code and content is best. However, in some build environments, such as building in the cloud, iLoks cannot be used. To address this problem, PACE has created an internet service that will do code signing in place of a signing iLok. This signing service is sometimes referred to as Cloud Signing or the Cloud Signing Service. This service should not be confused with iLok Cloud, which is PACE's internet-based licensing system. The --allowsigningservice option enables wraptool to use the Cloud Signing Service. ILoks, if connected to the computer running wraptool, are always favored for code signing; so be sure to unplug your iLoks if you want to try out the Cloud Signing Service. Also, for this option to work, your build environment requires an Internet connection. Additionally, you need to specify an account and an account password to a user that has the privileges to sign for the publisher specified in the wrapconfig.

--localonly
If the operation requires data that must come from either the server or the wrapper cache and if you have all of the wrapper data that you need in your cache, then you can use this option, to do the operation without ever contacting the server. In this case, you don't need to specify an account, and Wraptool won't even check with the server to see if you have the correct/latest wrapper data. Be aware that you won't get any changes made on the server if you use this option.

--in MyInput
This is the primary input for the operation. Typically, MyInput is a file or directory path that you must provide, but it may be something else for some operations.

--out MyOutput
This is the primary output for the operation. MyOutput is a file or directory path that you must provide, so that Wraptool can save data in this location for you.

--wcguid MyWrapConfigGuid
--wcfile MyWrapConfigFilePath
You can reference a Wrap Config in two ways. Choose one or the other, not both. If you want to use a Wrap Config from the server or the wrapper cache, then use --wcguid, and provide the GUID of the Wrap Config. Wrap Config GUIDs can be found on the server, and are shown in some Wraptool operations. If you have exported the Wrap Config to a file, then you can use --wcfile with the path of the file. If you specify a file, then the server will not be contacted about providing a Wrap Config.

--binaries MyWrapperBinariesDirectoryPath
--installedbinaries

These override the default wrapper binaries that Wraptool would have used; so it is best not to use these arguments unless you really understand what you are doing. If the operation uses a Wrap Config, then the Wrap Config indicates the correct wrapper binaries to use. If the operation has no Wrap Config, then the binaries installed with WrapTool are used. If you decide to override the default binaries, then you must choose only one of these arguments. The --binaries argument allows you to provide the path to the directory that contains the binaries the you want to use for any operation. The --installedbinaries argument causes Wraptool to override the Wrap Config settings and use the binaries installed with WrapTool.

Argument Spaces

Spaces and even some special symbol characters in names and paths can cause WrapTool to misinterpret the arguments. Be sure to either use quotes:

```
   --wcfile "My Wrap Config.wcg"
```

or backslashes

```
   --wcfile My\ Wrap\ Config.wcg
```

as appropriate for your computer platform.

wraptool help or wraptool -h

This operation displays some help information about how to use Wraptool.

```
   Examples:
      wraptool help
      wraptool -h
```

wraptool clean

This operation cleans out your wrapper cache, which contains Wrapper Binaries, Wrap Configs and Experiences.

```
   Optional Common Arguments:
         --verbose

   Examples:
      wraptool clean
      wraptool clean --verbose
```

wraptool sync

This operation updates the wrapper data in your wrapper cache to match the data on the server.

```
   Required Common Arguments:
         --account MyUserAccountName

   Optional Common Arguments:
         --password MyPassword
```

```
        --force
        --verbose

    Optional Publisher Identifier:
        If your user account is associated with multiple publishers. Specifying the --customernumber or --publisherid
        will limit the sync to the specified
            --customernumber MYCU-STOM-ERNU-MBER
            --publisherid MyPublisherId (Coming soon. Not implemented in WrapTool yet.)

    Examples:
        wraptool sync --account MyUserAccountName --password MyPassword --verbose
        wraptool sync --account MyUserAccountName --password MyPassword --verbose --customernumber MYCU-STOM-ERNU-MBER
```

wraptool list

This operation displays a list of the Wrapper Binaries, Wrap Configs and Experiences in your wrapper cache.

```
    Optional Common Arguments:
        --verbose

    Examples:
        wraptool list
        wraptool list --verbose
```

wraptool wrap

This operation applies the wrapper to an executable and optionally encrypts and/or digitally signs the executable. If you just want to sign the executable then use the sign operation. Signing an executable normally needs an iLok2 or iLok3 with a code/content signing certificate. Users can use the ILok License Manager App to get a code/content signing certificate, just by right clicking on the iLok icon and selecting "Synchronize" from the popup menu. Only users with the appropriate privileges will get a code/content signing certificate. If a user is associated with multiple publishers, then the certificate will contain the credentials to sign code/content for any of those publishers.

For performance reasons, using iLoks for signing code and content is best. However, in some build environments, such as building in the cloud, iLoks cannot be used. To address this problem, PACE has created an internet service that will do code signing in place of a signing iLok. This signing service is sometimes referred to as Cloud Signing or the Cloud Signing Service. This service should not be confused with iLok Cloud, which is PACE's internet-based licensing system.

The --allowsigningservice option enables wraptool to use the Cloud Signing Service. ILoks, if connected to the computer running wraptool, are always favored for code signing; so be sure to unplug your iLoks if you want to try out the Cloud Signing Service. Also, for this option to work, your build environment requires an Internet connection. Additionally, you need to specify an account and an account password to a user that has the privileges to sign for the publisher specified in the wrapconfig.

Required: The Wrap Config:

You need to specify one of these to indicate the Wrap Config to use:

```
--wcguid ED052BE6-9B79-4E72-A838-E226320B778E
--wcfile MyWrapConfigFile.wcg
```

Optional Common Arguments:

```
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
--force
--localonly
--allowsigningservice
--verbose
```

Required: The Input Executable:

You need to specify the path of the executable to wrap.

```
--in MyExecutableToWrapFilePath
```

Required: The Output Executable:

You need to specify the path to save the protected executable. If a file exists at the given path, then it will be replaced:

```
--out MyWrappedExecutableFilePath
```

Windows Note:

On Windows, the --in and --out arguments must either refer to a binary executable (such as a '.exe' or '.dll' file) or to the top level directory of an AAX plugin package. In the case of an AAX package, plugin DLLs with the extension '.aaxplugin' stored under the package sub-directories "Contents\Win32" and "Contents\x64" will be wrapped. The same path to the AAX package directory must be passed for both the --in and --out arguments. Even though the AAX package is a directory, it's path must be passed on the command line without a trailing '\' or a wraptool error may occur.

Required If Signing: The Signing Arguments:

Some of these arguments are needed if you are signing your executable. The Wrap Config defines if the executable should be signed or not; but this can be overridden with the --dsig argument.

```
--dsig on
--dsig off
```

Each PACE signed executable uses both a platform specific signature and an iLok2 or iLok3 signature. Wraptool will automatically sign using the iLok2 or 3 as long as the iLok is connected and the license support software is installed. Note that the license support software is automatically installed with the Eden SDK.

For Macintosh platform signing, the --signid argument is required to identify the certificate chain and private key in your keychain. Usually this signing ID should be an Apple provided application developer ID.

```
--signid "Developer ID Application: My Company Name"
```

Normally, all keychains that are on the calling user's keychain search list will be searched for the signing identity. However, you can optionally specify a specific keychain file to search by using the --keychain argument. This argument is passed directly to the codesign tool, and behaves as documented in the codesign “man” page.

```
--keychain MyKeychainFile
```

For signing on the Windows platform, a PKCS12 formatted key file can be used. Usually the key file should contain an Authenticode certificate from an approved Microsoft provider. A PKCS12 formatted key file can be used directly and is specified using --keyfile argument.

```
--keyfile MyKeyFilePath
```

Windows key files optionally have passwords. If your key file has a password, use the --keypassword argument to specify this password when --keyfile argument is also used.

```
--keypassword MyKeyFilePassword
```

However, putting the password in a command line script would not be very secure. So all you have to do is present your password once, and the password will be automatically captured in the secure storage of your computer. Once done, you don't need to provide this argument anymore unless you change your password. Logging into your computer is what unlocks access to this password in the secure storage of your computer.

Another way to sign under Windows is to provide wraptool with the reference to the code signing certificate located in your “Personal” certificate store. To do this, you must import the key file to the Personal certificate store of the current user, get the SHA1 thumbprint (string of 40 characters) of the certificate, then use that thumbprint as an identifier with –-signid argument.

```
--signid "8805b359edea7e86026f2c7a33d9ee297db191de"
```

At the time of this writing, “Extended Validation” certificates require a hardware dongle from a manufacturer other than PACE. If you use an EV certificate, you must follow the certificate provider’s instructions for installing the certificate in your personal certificate store, then you must use the above --signid option to reference the EV certificate. Under the hood wraptool will call signtool.exe, which in turn knows how to use the EV certificate dongle to provide the Windows level code signature. Please refer to the documentation from your EV certificate provider for details regarding installing and using your EV certificate and dongle.

The platform signing tool itself can have its own options. The --extrasigningoptions argument is used to specify these options. Be sure the quote or use backslashes to deal with spaces in the options.

```
--extrasigningoptions "option1 option2 option3"
--extrasigningoptions option1\ option2\ option3
```

A typical use of the --extrasigningoptions argument is to provide signing requirements to the Mac codesign tool. Here’s an example of how to do this:

```
--extrasigningoptions '--requirements "PATH_TO_YOUR_REQUIREMENTS_FILE"'
```

When digitally signing wraptool automatically retries if the platform's timestamp server is unavailable. The goal of this change is to help deal with occasional temporary Internet connectivity problems or timestamp server outages that would otherwise halt a build.

By default wraptool will retry for up to 10 minutes with a 10 second delay between attempts. You can change these values via the new --timestampretry and --timestampretrysleep options.

Apple Notarization options:

If you wish to notarize your signed binary, you will need to set up the notarization prerequisites required by Apple. Then at a minimum you must provide the following arguments to wraptool:

```
--notarize-username YOUR_APPLE_ID
--notarize-password @keychain:APP_PASSWORD_ITEM_NAME
```

See the “macOS Notarization Support” section of the GettingStartedGuide.html for details on setting up notarization and using the above options.

In addition to the above required arguments, there are also some optional related notarization arguments. If you are a member of multiple developer teams, you may need to provide this option when notarizing in order to specify which team to use for the notarization operation:

```
--notarize-ascprovider YOUR_ASC_PROVIDER
```

If you don’t need to notarize your binary directly but plan to include it in an installer package or disk image that will be notarized, then you must sign your binary with the digital signature options required by Apple for notarization. The following wraptool option will handle setting the appropriate codesign options for you:

```
--dsigharden
```

Mac OS Specific Issues:

As part of installation of the wrapper, wraptool has to add new Mach-O structures to the target binary. Due to architectural limitations in the Mach-O format, there is a fixed amount of space where Mach-O load commands can be added to a binary. As a result, some binaries will not have sufficient header pad to allow wrapping, which will cause a wrap time error.

Digital signing also requires a small amount of header pad, but indirectly. Since wraptool uses Apple’s codesign tool, and that tool also has header pad requirements, it’s possible to run into situations where the codesign tool itself will fail.

Whether or not a binary has enough header pad to be wrapped or signed is somewhat random. You might have a binary that wraps fine today, but after a few changes and a relink, the same product cannot be wrapped because a linker threshold been crossed.

To avoid this problem, you should add the “-headerpad” option in the “Other linker flags” to your projects in order to ensure that there will always be enough header pad to wrap or sign your binaries:

```
OTHER_LDFLAGS = -Xlinker -headerpad -Xlinker 578
```

If you use Xcode xcconfig files, you might want to globally define this linker option for all of your projects.

Optional: Wrap Config Overrides:

Warning: These overrides are meant for temporary testing situations. Overriding Wrap Config settings on a permanent basis is not recommended, unless the settings force a more secure result. In any case, using these overrides will likely cause confusion and unintentionally wrong protection settings in your wrapped executables.

Binary signing adds to the security of your executable and is highly recommended. The binary signing option can be overridden as explained above in the signing arguments section.

The Experience specified in the Wrap Config can be overridden by providing the path to either the experience zip file or the directory of an expanded experience.

```
--experience MyExperienceZipOrDirectoryPath
```

Stripping debug symbols is important for the security of executables that you release. The strip debug symbols option specified in the Wrap Config can be forced on or off by the following arguments:

```
--strip on
--strip off
```

Not allowing debugging is important for the security of executables that you release. The allow debugging option specified in the Wrap Config can be forced on or off by the following arguments:

```
--allowdb on
--allowdb off
```

Fusion adds to the security of your executable and is highly recommended. The fusion option specified in the Wrap Config allows Wraptool to double-check that you have fusion protected your executable prior to wrapping.
This fusion check can be forced on or off by the following arguments:

```
--fusion on
--fusion off
```

There are two fusion types, classic and DLC. DLC is the newer version and requires an iLok firmware update. If you have fusion on and you are using DLC, it is important to have the DLC option enabled in the wrapper. When the fusion and DLC options are both on, then the wrapper will verify that the iLok has the the firmware version to support DLC. If the firmware is not new enough, the Experience will launch and update the firmware automatically. Without these options enabled, no automatic firmware update will happen. Instead, users will be instructed how to manually update the iLok firmware when a DLC fusion check fires off and fails.
This DLC option can be forced on to indicate DLC fusion or off to indicate classic fusion by the following arguments:

```
--dlc on
--dlc off
```

The beta expiration option specified in the Wrap Config allows an executable to stop working after the beta expiration date even if the user has a valid auth for this executable. The beta expiration option can be forced on, off or set on and to a different date by the following arguments. The date/time value is expressed in the ISO 8601 form.

```
--beta on
--beta off
--beta 2013-04-20T00:00:00
```

The wrapconfig's valid locations setting, which defines the locations that will be searched for licenses, can be overridden. There are 16 valid locations settings to choose from:

Valid Location Setting	Locations
localOnlyIlok2OrGreater	Ilok2, Ilok3
localOnlyIlok	Ilok1, Ilok2, Ilok3
localOnlyAllButIlok1	Ilok2, Ilok3, LicenseDb
localOnlyAll	Ilok1, Ilok2, Ilok3, kLicenseDb
remoteOrLocalIlok2OrGreater	Ilok2, Ilok3, Ilok2Remote, kIlok3Remote
remoteOrLocalIlok	Ilok1, Ilok2, Ilok3, Ilok2Remote, Ilok3Remote
remoteOrLocalAllButIlok1	Ilok2, Ilok3, LicenseDb, Ilok2Remote, Ilok3Remote, IlokCloudRemote
remoteOrLocalAll	Ilok1, Ilok2, Ilok3, kLicenseDb, Ilok2Remote, Ilok3Remote, IlokCloudRemote
localModernIlokWithCloud	Ilok2, Ilok3, IlokCloudRemote
localModernLocationWithCloud	Ilok2, Ilok3, LicenseDb, IlokCloudRemote
remoteOrLocalModernIlokWithCloud	Ilok2, Ilok3, Ilok2Remote, Ilok3Remote, IlokCloudRemote
localIlokWithCloud	Ilok1, Ilok2, Ilok3, IlokCloudRemote
localLocationWithCloud	Ilok1, Ilok2, Ilok3, LicenseDb, IlokCloudRemote
remoteOrLocalIlokWithCloud	Ilok1, Ilok2, Ilok3, Ilok2Remote, Ilok3Remote, IlokCloudRemote
remoteOrLocalAllButIlok1AndCloud	Ilok2, Ilok3, LicenseDb, Ilok2Remote, Ilok3Remote
remoteOrLocalAllButCloud	Ilok1, Ilok2, Ilok3, LicenseDb, Ilok2Remote, Ilok3Remote

Local means the license must be on a location directly attached to the computer where the software is being run. Remote means that location with the license can be on a PACE remote license server, which is accessed via a network.

```
--validlocations remoteOrLocalAll
```

Optional: Miscellaneous Options:

Date/time values are provided to WrapTool are assume to be expressed in the computer's local time zone. By providing this argument, the dates will be interpreted in UTC time (also known as GMT or Greenwich Mean Time):

```
--utc
```

Wraptool does not wrap Mac PPC executables. If your universal binary is built with PPC you will get an error when you try to wrap. Wraptool will strip out unsupported architectures and wrap the remaining architectures in the protected output executable if you provide this argument:

--removeunsupportedarchs

Examples:

```
wraptool wrap --wcguid ED052BE6-9B79-4E72-A838-E226320B778E --account MyUserAccountName
              --in MyExecutableToWrapFilePath --out MyWrappedExecutableFilePath

wraptool wrap --wcfile MyWrapConfigFile.wcg --in MyExecutableToWrapFilePath
              --out MyWrappedExecutableFilePath

wraptool wrap --wcfile MyWrapConfigFile.wcg --in MyExecutableToWrapFilePath
              --out MyWrappedExecutableFilePath --signid "Developer ID Application: My Company Name"
```

Optional: InterLok Adapter:

WARNING! The InterLok Adapter feature has been deprecated. You may continue to wrap with this feature for the time being, but with limitations. See the release notes or the PaceEdenInterLokCompatibility.pdf document for details.

This Macintosh-only option is available for customers who wish to create wrapped binaries, but have them use InterLok for the license validation instead of Eden. Reasons to do this include the need to create 64 bit wrapped binaries before Eden has shipped.

To implement this option in your products, you must do the following:

1. Install InterLok for Macintosh v5.9.1 or greater, as needed. Previous versions of InterLok will not work correctly.
2. Use the installed 5.9.1 MasterMaker to protect this shared library (saving the protected copy to somewhere other than the installed location):

   /Applications/PACEAntiPiracy/Eden/Fusion/Legacy/WrapThisWithInterLok.bundle

3. On PACE Central 2, set up (as needed) a wrap config that matches the product and options in the options document used to protect the “WrapThisWithInterLok.bundle” bundle. You must specify this wrap config, either by GUID or file, when wrapping your product.

4. Use wraptool to wrap a binary, specifying the InterLok protected bundle created in step 2 using the new "--interlokadapter" option, and the wrap config created in step 3.

   Here's a sample command line (you must substitute your account ID, binary to protect, adapter binary, and wrap config GUID):

   /Applications/PACEAntiPiracy/Eden/Fusion/Versions/2/bin/wraptool wrap --verbose --account my_ilok_account --in Sample.app --out Sample_wrapped.app --interlokadapter WrapThisWithInterLok_prot.bundle --wcguid E29A23B3-AE57-4E6F-BBEB-71A7F5798BB4

The resulting binary will be wrapped with Eden, but the authorization will be validated by the InterLok "adapter" provided in the InterLok wrapped shared library. Most of the usual Eden wrap config options directly apply to the wrapped binary, like digital signing, symbol striping, code encryption, beta expiration, and what to do when unauthorized. But any options related to licensing (like valid license locations) are ignored since those are under the control of the InterLok wrapped adapter.

If you run your protected binary when unauthorized, the usual InterLok Client Authorization Wizard (or CAW) will be displayed. If supported, InterLok Activation can be used to activate the user.

Your wrapped software will have no dependencies on Eden end user software. You only need to provide your customers with the usual InterLok Extensions for them to use your protected software. If you include the extensions in your installer, make sure to use the 5.9.1 extensions or greater. Otherwise your customers can simply download the InterLok extensions from PACE.

WARNING! The Eden wrapper has different system requirements than InterLok. You must set your deployment target to 10.5 or above.

wraptool info

This operation displays the wrapper info in a wrapped executable.

Required: The Input Executable:
You need to specify the path of the executable from which to get the wrapper info.
--in MyWrappedExecutableFilePath

Optional Common Arguments:
--account MyUserAccountName
--password MyDeveloperPassword

```
        --binaries MyWrapperBinariesDirectoryPath
        --installedbinaries
        --force
        --localonly
        --verbose
```

wraptool simulate

Simulates the wrapper auth verification and activation experience without having to wrap an executable.

Required: The Wrap Config:

You need to specify one of these to indicate the Wrap Config to use:

```
        --wcguid ED052BE6-9B79-4E72-A838-E226320B778E
        --wcfile MyWrapConfigFile.wcg
```

Optional Common Arguments:

```
        --account MyUserAccountName
        --password MyDeveloperPassword
        --binaries MyWrapperBinariesDirectoryPath
        --installedbinaries
        --force
        --localonly
        --verbose
```

Optional: Wrap Config Overrides:

The Experience specified in the Wrap Config can be overridden by providing the path to either the experience zip file or the directory of an expanded experience.

```
        --experience MyExperienceZipOrDirectoryPath
```

The beta expiration option specified in the Wrap Config can be forced on, off or set on and to a different date by the following arguments. The date/time value is expressed in the ISO 8601 form.

```
        --beta on
        --beta off
        --beta 2013-04-20T00:00:00
```

The wrapconfig's valid locations setting, which defines the locations that will be searched for licenses, can be overridden. There are eight valid locations settings to choose from: localOnlyIlok2OrGreater, localOnlyIlok,

localOnlyAllButIlok1, localOnlyAll, remoteOrLocalIlok2OrGreater, remoteOrLocalIlok, remoteOrLocalAllButIlok1 or remoteOrLocalAll. Local means the license must be on a location directly attached to the computer where the software is being run. Remote means that location with the license can be on a PACE remote license server, which is accessed via a network.
--validlocations remoteOrLocalAll

Examples:
wraptool simulate --wcguid ED052BE6-9B79-4E72-A838-E226320B778E --account MyUserAccountName
wraptool simulate --wcfile MyWrapConfigFile.wcg --experience MyExperienceZipOrDirectoryPath

wraptool sign

This operation just signs the executable without adding a wrapper. The Pace Eden Interface in the fusion library can be used to verify the signature. Signing an executable normally needs an iLok2 or iLok3 with a code/content signing certificate. Users can use the ILok License Manager App to get a code/content signing certificate, just by right clicking on the iLok icon and selecting "Synchronize" from the popup menu. Only users with the appropriate privileges will get a code/content signing certificate. If a user is associated with multiple publishers, then the certificate will contain the credentials to sign code/content for any of those publishers.

For performance reasons, using iLoks for signing code and content is best. However, in some build environments, such as building in the cloud, iLoks cannot be used. To address this problem, PACE has created an internet service that will do code signing in place of a signing iLok. This signing service is sometimes referred to as Cloud Signing or the Cloud Signing Service. This service should not be confused with iLok Cloud, which is PACE's internet-based licensing system. The --allowsigningservice option enables wraptool to use the Cloud Signing Service. ILoks, if connected to the computer running wraptool, are always favored for code signing; so be sure to unplug your iLoks if you want to try out the Cloud Signing Service. Also, for this option to work, your build environment requires an Internet connection. Additionally, you need to specify an account and an account password to a user that has the privileges to sign for the publisher specified in the wrapconfig.

Required: The Wrap Config or Some Basic Publisher Info:
You need to specify one of these to indicate the Wrap Config to use:
--wcguid ED052BE6-9B79-4E72-A838-E226320B778E
--wcfile MyWrapConfigFile.wcg

Or, you can instead provide:
--customernumber MYCU-STOM-ERNU-MBER

```
      --customername MyCompanyName
      --productname MySignedProductName

   In the second option, you can omit --productname, and then The Input Executable file name will be used instead.
   Also instead of --customernumber you can use this argument instead:
      --publisherid MyPublisherId (Coming soon. Not implemented in WrapTool yet.)


Optional Common Arguments:
      --account MyUserAccountName
      --password MyDeveloperPassword
      --binaries MyWrapperBinariesDirectoryPath
      --installedbinaries
      --force
      --localonly
      --allowsigningservice
      --verbose

Required: The Input Executable:
   You need to specify the path of the executable to sign.
      --in MyExecutableToSignFilePath

Required: The Output Executable:
   You need to specify the path to save the signed executable. If a file exists at the given path, then it will be
   replaced:
      --out MySignedExecutableFilePath

Windows Note:
   On Windows, the --in and --out arguments must either refer to a binary executable (such as a '.exe' or '.dll'
   file) or to the top level directory of an AAX plugin package. In the case of an AAX package, plugin DLLs with the
   extension '.aaxplugin' stored under the package sub-directories "Contents\Win32" and "Contents\x64" will be
   signed. The same path to the AAX package directory must be passed for both the --in and --out arguments. Even
   though the AAX package is a directory, it's path must be passed on the command line without a trailing '\' or a
   wraptool error may occur.

Required: The Signing Arguments:
   Each PACE signed executable uses both a platform specific signature and an and an iLok2 or iLok3 signature.
   Wraptool will automatically sign using the iLok2 or 3 as long as the iLok is connected and the license support
   software is installed. Note that the license support software is automatically installed with the Eden SDK.
```

For Macintosh platform signing, the --signid argument is required to identify the certificate chain and private key in your keychain. Usually this signing ID should be an Apple provided application developer ID.

```
--signid "Developer ID Application: My Company Name"
```

Normally, all keychains that are on the calling user's keychain search list will be searched for the signing identity. However, you can optionally specify a specific keychain file to search by using the --keychain argument. This argument is passed directly to the codesign tool, and behaves as documented in the codesign “man” page.

```
--keychain MyKeychainFile
```

For signing on the Windows platform, a PKCS12 formatted key file can be used. Usually the key file should contain an Authenticode certificate from an approved Microsoft provider. A PKCS12 formatted key file can be used directly and is specified using --keyfile argument.

```
--keyfile MyKeyFilePath
```

Windows key files optionally have passwords. If your key file has a password, use the --keypassword argument to specify this password when --keyfile argument is also used.

```
--keypassword MyKeyFilePassword
```

However, putting the password in a command line script would not be very secure. So all you have to do is present your password once, and the password will be automatically captured in the secure storage of your computer. Once done, you don't need to provide this argument anymore unless you change your password. Logging into your computer is what unlocks access to this password in the secure storage of your computer.

Another way to sign under Windows is to provide wraptool with the reference to the code signing certificate located in your “Personal” certificate store. To do this, you must import the key file to the Personal certificate store of the current user, get the SHA1 thumbprint (string of 40 characters) of the certificate, then use that thumbprint as an identifier with –-signid argument.

```
--signid "8805b359edea7e86026f2c7a33d9ee297db191de"
```

At the time of this writing, “Extended Validation” certificates require a hardware dongle from a manufacturer other than PACE. If you use an EV certificate, you must follow the certificate provider’s instructions for installing the certificate in your personal certificate store, then you must use the above --signid option to reference the EV certificate. Under the hood wraptool will call signtool.exe, which in turn knows how to use the

EV certificate dongle to provide the Windows level code signature. Please refer to the documentation from your EV certificate provider for details regarding installing and using your EV certificate and dongle.

The platform signing tool itself can have its own options. The --extrasigningoptions argument is used to specify these options. Be sure the quote or use backslashes to deal with spaces in the options.

```
--extrasigningoptions "option1 option2 option3"
--extrasigningoptions option1\ option2\ option3
```

A typical use of the --extrasigningoptions argument is to provide signing requirements to the Mac codesign tool. Here's an example of how to do this:

```
--extrasigningoptions '--requirements "PATH_TO_YOUR_REQUIREMENTS_FILE"'
```

When digitally signing wraptool automatically retries if the platform's timestamp server is unavailable. The goal of this change is to help deal with occasional temporary Internet connectivity problems or timestamp server outages that would otherwise halt a build.

By default wraptool will retry for up to 10 minutes with a 10 second delay between attempts. You can change these values via the --timestampretry and --timestampretrysleep options.

Examples:

```
wraptool sign --wcguid ED052BE6-9B79-4E72-A838-E226320B778E --account MyUserAccountName
              --signid "Developer ID Application: My Company Name"

wraptool sign --customernumber MYCU-STOM-ERNU-MBER --customername MyCompanyName
              --productname MySignedProductName --keyfile MyKeyFilePath
```

Apple Notarization options:

If you wish to notarize your signed binary, you will need to set up the notarization prerequisites required by Apple. Then at a minimum you must provide the following arguments to wraptool:

```
--notarize-username YOUR_APPLE_ID
--notarize-password @keychain:APP_PASSWORD_ITEM_NAME
```

See the "macOS Notarization Support" section of the GettingStartedGuide.html for details on setting up notarization and using the above options.

In addition to the above required arguments, there are also some optional related notarization arguments. If you are a member of multiple developer teams, you may need to provide this option when notarizing in order to specify which team to use for the notarization operation:

```
--notarize-ascprovider YOUR_ASC_PROVIDER
```

If you don't need to notarize your binary directly but plan to include it in an installer package or disk image that will be notarized, then you must sign your binary with the digital signature options required by Apple for notarization. The following wraptool option will handle setting the appropriate codesign options for you:

```
--dsigharden
```

Mac OS Specific Issues:

Signing, like wrapping but to a lesser degree, requires some free space in the target binary's Mach-O header. This subject is discussed in detail in the "wraptool wrap" section under "Mac OS Specific Issues".

Without going into further detail and duplication here, you may need to force the header pad of your binaries to be sufficient for signing or wrapping. To do so, you need to set the following option in your projects or xcconfig files:

```
OTHER_LDFLAGS = -Xlinker -headerpad -Xlinker 578
```

wraptool verify

This operation verifies the digital signature of an executable and displays the verification results and signature info if the signature if present and valid.

Required: The Input Executable:

You need to specify the path of the executable to verify.

```
--in MyExecutableToVerifyFilePath
```

Optional Common Arguments:

```
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
--force
--localonly
--verbose
```

wraptool dump

Displays the contents of a Wrap Config. Some items in the WrapConfig will be omitted from the display, because of security reasons.

Required: The Wrap Config:
You need to specify one of these to indicate the Wrap Config to use:
--wcguid ED052BE6-9B79-4E72-A838-E226320B778E
--wcfile MyWrapConfigFile.wcg

wraptool export

This operation exports the target Wrap Config to a Wrap Config file and/or exports the experience resources associated with the target Wrap Config.

Optional: The Wrap Config:
If you want to export the Wrap Config, the you need to specify one of these to indicate the Wrap Config to use:
--wcguid ED052BE6-9B79-4E72-A838-E226320B778E
--wcfile MyWrapConfigFile.wcg

Optional Common Arguments:
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
--force
--localonly
--verbose

Optional: A path to save the Wrap Config file:
Export will always add or replace the file extension specified with ".wcg", the file extension for Wrap Config files. Wrap Config files, which contain product specific secrets, are always encrypted, but care should still be taken to keep these files safe.

--out MyWrapConfig.wcg

Optional: A path to save the Experience Resources as a standard zip file:

```
    Export will always add or replace the file extension specified with ".zip". Experience resources do not contain
    any security sensitive information.

        --experience MyExperience.zip

Examples:
    wraptool export --wcguid ED052BE6-9B79-4E72-A838-E226320B778E --account MyUserAccountName
                    --out /MyPath/MyWrapConfig.wcg

    wraptool export --wcguid ED052BE6-9B79-4E72-A838-E226320B778E --account MyUserAccountName
                    --out /MyPath/MyWrapConfig.wcg --experience /MyPath/MyExperienceResources.zip
```

wraptool codegen

This operation generates license data in the form of C code. The data defined in this code file is really just encrypted data that the PACE Eden interface needs to find and verify the auths allowed by the Wrap Config that you specify in this command. The license data can be incorporated into code that uses the PACE Eden interface. Alternatively, you could include the license data along with the corresponding protected content. For an overview of the content protection, please see the "PaceEdenContentProtection.pdf" document, which is included in the PACE Eden SDK.

```
    Required: The Wrap Config:
        You need to specify one of these to indicate the Wrap Config to use:
            --wcguid ED052BE6-9B79-4E72-A838-E226320B778E
            --wcfile MyWrapConfigFile.wcg

    Optional Common Arguments:
            --account MyUserAccountName
            --password MyDeveloperPassword
            --binaries MyWrapperBinariesDirectoryPath
            --installedbinaries
            --force
            --localonly
            --verbose

    Required: A path to save the PACE Eden license data as a C code text file:
        This contains the license data that needs to be available to the PACE Eden Authorization and Content Functions
        when the a protected product is used. If a file already exists at the given path, then this file will be
        replaced.
```

```
    --out PaceEdenLicenseData.cpp
```

Optional: A license data common key to be incorporated into the license data for the purposes of decrypting content:

If you want to encrypt content, then you must provide the license data common key, so that the appropriate information is created in the Pace Eden license data. The PACE Eden Interface supports 128-bit AES content encryption. Two types of keys, the license data common key and the product specific keys are used in content encryption. The product specific keys are only relevant if your Wrap Config defines multiple auths that can authorize your product. The license data common key is always accessible from the license data if any auth defined in the Wrap Config is present. If you want either auth A or auth B to authorize product C, then the Wrap Config for product C would have to contain both auth A and auth B definitions. It may be desirable to have some premium content that can only be decrypted when auth A is present, because auth A represents a premium product purchase. In this case, the product specific key for auth A would be used to encrypt the premium content.

As the content publisher, you must specify the license data common key. For every auth product defined in the Wrap Config, a product specific key is then generated from the license data common key. All of these keys are then stored in encrypted form in the license data and the content key file discussed below. Product specific keys are always generated consistently with the auth definitions, so that the same license data common key always generates the same product specific keys for the same auth definitions regardless of what Wrap Config is used.

There are various ways to specify a license data common key:

Provide a path to a content key file that was previously generated by codegen:

```
    --inkey MyExistingContentKeyFile.cke
```

OR randomize a new license data common key and initialization vector by not specifying the inkey option, but at the same time using the outkey option described below to save the random key.

OR provide the license data common key and initialization vector as a 64-digit hex number as shown below, where the first 32 digits is the key and the last 32 digits is the initialization vector. You can omit the initialization vector, which will result in an all zero initialization vector. The two examples below are effectively the same.

```
    --inkey 0x1234567890ABCDEF1234567890ABCDEF00000000000000000000000000000000
    --inkey 0x1234567890ABCDEF1234567890ABCDEF
```

OR if you don't want to encrypt content and use the license data with an auth to decrypt content, then omit both the inkey and outkey options. This will make the license data a bit smaller too.

Optional: A path to save the content key file that is based on the provided license data common key:

Providing this option with a path will cause a content key file to be generated. This operation will always add or replace the file extension specified with ".cke", the file extension for key files. A content key file is required to do content encryption. The content key file will be encrypted, but callers should keep this file in a safe place and should never ship this file with protected products. The content key file will contain the license data common key plus a product specific key for each auth product defined in the Wrap Config. The product specific keys are generated consistently from the license data common key, so that the same license data common key will always generate the same product specific key from the same auth definition regardless of the Wrap Config. This gives the caller the ability to maintain the same keys across different Wrap Configs. However, if the set of keys needs to be changed, then the caller can accomplish this be using another license data common key and generating a new content key file. As for the initialization vector, all of the keys in the content key file use the same initialization vector.

Here is how to specify the path for the output content key file. If a file already exists at the given path, then this file will be replaced.

```
--outkey MyNewContentKeyFile.cke
```

IMPORTANT: If you are specifying or randomizing a new key as described in the inkey option above, then you need the provide the outkey option, so you can encrypt using this new key. The license data has been configured to generate the same keys as the key file when the license data is in the presence of an auth; so the caller must keep the license data, the key file and content files organized, so that there is no key mismatch between these items. Be careful when randomizing a new key, because that is when a mismatch between the license data, the key file and content files usually happens. If you are having trouble decrypting newly encrypted content, this sort of mismatch is usually to blame.

If you are providing a previously generated content key file in the inkey, should you also specify the outkey? It depends on the situation:

If you are not using any product specific keys, then you don't have to used the outkey option, since you are just using the same license data common key that is in the input file.

If you are using a product specific key and if you are just regenerating the license data, which is good to do for every product release, and if you are using a Wrap Config with the same set of defined auths used when the content key file was previously generated, then there is no need to use the outkey option, since the same content key file will be generated.

If you are using a product specific key and if you are using a Wrap Config with a different set of auth products used when the content key file was previously generated, then you should use the outkey option to regenerate the content key file with the new set of keys for the new set of auth products.

Optional: Override allowed PlayerIds used in Player Validation for content decryption:

By default, player validation is enabled in the generated license data, whenever a content key is provided in this codegen operation. Player validation will occur when verifyAuth is called in the PACE Eden API and when the caller is requesting the ContentDecryptKey structure. The ContentDecryptKey structure is required to decrypt any content encrypted with the key used in this WrapTool codegen operation. Player validation involves validating the signature of the player's executable, meaning the player has to be signed using the WrapTool sign operation. Furthermore, the publisherId from the signature of the player must be in the list of allowed playerIds that is embedded in the license data.

By default, the allowed playerIds are the publisherIds from all of the auth products defined in the Wrap Config. Using this option allows the caller to override the default allowed playerIds. The playerIds can be specified in decimal or hex. Here are some examples of overriding the default allowed playerIds with just one playerId:

```
--allowedplayerids 20
--allowedplayerids 0x14
--allowedplayerids "0x14"
```

You can specify multiple playerIds separated by spaces; however quotes are required in this case. Here are some examples:

```
--allowedplayerids "1 2 20"
--allowedplayerids "0x1 0x2 0x14"
```

You can disable player validation completely. This is fine for testing, but is not recommended for released products, because without player validation other PACE publishers, using the PACE Eden API, can decrypt and use your content in the presence of an auth that authorizes your content. You can disable player validation by specifying this override option without any playerIds; however in this case, quotes are also required. Here is the example for disabling player validation:

```
--allowedplayerids ""
```

Optional: Override for valid locations:

The wrapconfig's valid locations setting, which defines the locations that will be searched for licenses, can be overridden. There are eight valid locations settings to choose from: localOnlyIlok2OrGreater, localOnlyIlok, localOnlyAllButIlok1, localOnlyAll, remoteOrLocalIlok2OrGreater, remoteOrLocalIlok, remoteOrLocalAllButIlok1 or remoteOrLocalAll. Local means the license must be on a location directly attached to the computer where the software is being run. Remote means that location with the license can be on a PACE remote license server, which is accessed via a network.

```
--validlocations remoteOrLocalAll
```

Optional: The type of license data file:

You can have the license data exported to a C code text file (the default) or a standard binary data file. The advantages of the C code text file is that it contains comments about the license data settings and is more convenient to use with source control systems. The disadvantage of the C code text file is that the file sizes will be bigger than a standard binary data file.

Provide this option to export the license data as a standard binary data file:

```
--exportbinary
```

OR don't provide this option in order to export the license data as a text file containing C code.

wraptool dcgen

This operation generates source code for the Dynamic Crypto feature. Dynamic Crypto allows your products to encrypt and decrypt data at runtime, but only in the presence of one or more valid licenses. Dynamic Crypto is different than our content protection feature in that you can dynamically encrypt and decrypt end-user data at runtime in your products, rather than just decrypting static content that was previously created by wraptool.

Dynamic Crypto uses cryptographically strong algorithms implemented in a series of generated whitebox instances. The generated whitebox code is highly obfuscated and resistant to reverse engineering and key recovery attacks.

Since Dynamic Crypto uses this highly attack resistant whitebox technique to perform the cryptographic operations, it performs more slowly than typically available cipher implementations that are less secure. Additionally, since a license is required periodically (roughly every 4.25 minutes) in order to perform encryption or decryption, there could be a performance hit when a license is obtained.

For these reasons, you should consider using Dynamic Crypto in situations where performance is not critical. The actual performance of Dynamic Crypto can be seen during the testing phase, where a test application runs and reports the block encryption speeds to the developer.

For a demonstration of Dynamic Crypto integration, please see the ThunderChicken sample. ThunderChicken uses Dynamic Crypto in the PACE protected targets to encrypt and decrypt audio settings data, so you'll see its effects when you try to save or load with no license present.

You can use Dynamic Crypto with all modern license locations, including iLok2, iLok3, host-based, and floating licenses. The only limitation is that Dynamic Crypto cannot work with iLok1 licenses. Like our other licensing features, you can use multiple licenses with Dynamic Crypto. If the wrap config used to generate the Dynamic Crypto sources contains multiple products, then any of the corresponding product licenses can be used by Dynamic Crypto to encrypt or decrypt data.

IMPORTANT! If you're using Dynamic Crypto and allow host-based licenses, your customers will need to install the latest License Support software. If a user is running a version of License Support that is older than 3.1.0 and their license is on their host machine (not an iLok), the DynamicCryptoManager will return an error indicating that host-based licenses are not supported. It is up to you to inform the user that they must update to the latest License Support version, or move their license to an iLok.

The Dynamic Crypto code generated by wraptool is cross platform. As a result, you can pick the platform you prefer, perform the code generation there, then use the generated code for all platforms. For example, you can generate the code on a Macintosh then use it in both your Mac and Windows projects.

Once your Dynamic Crypto code is generated and integrated into your product, you can optionally exercise it for Fusion protection. The result is that the generated sources will be injected for protection, adding even more strength against influence by attackers. Note that although there will be different code generated for 32 and 64 bit platforms, you can exercise on a single platform and the analysis data can be used for both.

IMPORTANT! The whitebox generated code used by Dynamic Crypto is not thread safe. If you're using Dynamic Crypto in a multithreaded environment, you must synchronize access to the encrypt and decrypt methods. Please see the comment documentation in the DynamicCryptoManager.h template or generated code for details.

Required: The Wrap Config:

You need to specify one of these to indicate the Wrap Config to use:

```
--wcguid ED052BE6-9B79-4E72-A838-E226320B778E
--wcfile MyWrapConfigFile.wcg
```

Optional Common Arguments:

```
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
```

```
--force
--localonly
--verbose
```

If you pass the --verbose option, then you'll be provided with the whole whitebox code generation output, as well as the build log for the test tool (if you haven't disabled testing). This can be a tremendous amount of information. If you don't use the --verbose option, you'll be provided with concise status information during the generation and testing process.

Required: A path to a directory to save the generated Dynamic Crypto source code:

You must tell wraptool where to put the generated Dynamic Crypto code. If the directory does not exist, wraptool will create it for you. Here's an example of how to specify the output folder:

```
--out MyProductDynamicCryptoDir
```

Required: A license for your product in order to test the generated code:

If you opt to have wraptool test your generated code (the default behavior), then you should make sure that you have a license for one of the products in your wrap config present at the time you're performing code generation. This way the license will be found and used by the test. Without the license present, the test will fail because Dynamic Crypto needs it to perform encrypt and decrypt operations.

Optional: Controlling the generated output:

If desired, you can control certain aspects of the generated code. First, you can affect the architecture that you want to support. The whitebox technology we're using needs to be aware of the pointer size. The result is that there will be different generated code for 32 and 64 bit architectures.

By default the dcgen operation will generate both the 32 and 64 bit code. But if you wish to only support a single architecture, you can specify the desired pointer size. For example, the following option will generate just the 64 bit version of the code:

```
--ptr-size 64
```

You may also specify a prefix string that will be prepended to the names of generated source files and to the names of symbols within those files. If you don't specify the prefix string, the generated code file and function names will be the default names from our Dynamic Crypto template source files.

Here's an example of specifying a prefix string:

```
--prefix "MyKillerApp"
```

Optional: Controlling the secrets used to create encryption keys:

The generated whitebox code uses secret keys in order to perform key exchanges with license locations. These secrets are effectively dissolved into the generated code itself such that it's very difficult for an attacker to identify and recover them. By default these secrets are obtained via a cryptographically secure algorithm from secret data associated with the top level product in your wrap config. In general we recommend that you use this default behavior and allow us to take care of the secret generation for you.

That having been said, there are some use cases where you might want to take responsibility for managing the secrets. For example, if you have encrypted data that is shared between multiple products, you may wish to come up with your own secret rather than let our tools derive it. This way you could use the same secret for code generation of multiple different products that have different wrap configs, and each product would be able to encrypt and decrypt the same user data in the presence of their respective licenses.

An advantage of this approach is that you're in complete control of the secrets, therefore you can arbitrarily match the secrets to whatever license you wish (even for products that have not yet been created). The downside of this approach is that you have to manage the secrets yourself, and you have to safely keep them somewhere in case you need to regenerate code in the future. If you lose your secrets, you will be unable to generate new instances of code that can decrypt user data in the field.

If you wish to use your own secrets, then you must initially generate them with a strong random number generator. Secret strings should be at least 32 bytes in length, unencoded. Here's a sample Mac OS X command line that will obtain 32 bytes of random data then encode it into uppercase hex:

```
od -vAn -N32 -tx1 < /dev/urandom | tr -d '\040\011\012\015'| awk '{print toupper($0)}'; echo
```

Once you have your secret, here's how to provide it to wraptool (use your own secret, of course):

```
--secret "E1B01DCE6780D6AD1B0EB07C3B02D21DA31BE7478B76E534F1D2EA8D0E7B377A"
```

Important! Don't forget to save your secrets and keep them secure. Don't ever lose them or you will not be able to generate new Dynamic Code instances that can decrypt your users' data. Also please be aware that secrets are case sensitive. You must preserve the exact secret string, case included.

If you initially opt for the default behavior of allowing wraptool to derive the secret from your wrap config's primary product, then you realize that you want to allow a new product with a different license to also access the same encrypted data, you can ask wraptool to tell you the derived secret via this option:

```
--showsecret
```

Once you have the secret string, you can pass it as the --secret option in future dcgen operations.

Optional: Affecting or varying whitebox code generation:

By default the whitebox tool will transmute the secret into a seed used to vary code generation. If you generate code with a product A wrap config, then you generate code with a product B wrap config, the generated code itself will be significantly different. You automatically get code generation variance due to the fact that the secrets are different. If you're managing your own secrets, then you get the same effect with different secrets.

But additionally we have the option for you to provide a seed that will further vary code generation. Reasons to do this might be because you want to release a new version of your product and you want the generated code to appear to be different, reseting any knowledge an attacker may have built up trying to reverse engineer your previous version.

If you're using Dynamic Crypto with a product for the first time, there really is no need to provide a custom seed. You might want to provide a seed in future major versions just to vary the generated code. But even this is not strictly necessary, especially if there's been no successful attack on your Dynamic Crypto implementation in earlier versions.

Unlike the option to provide your own secret, this seed value only varies the whitebox code generation. The resulting generated code will still have the same dissolved secrets and therefore will be able to decrypt user data in the field. The algorithm itself is not changed, but the implementation will appear to be dramatically different to an attacker.

Since this optional seed does not affect the secret keys, you don't have to keep it. You can randomly generate a seed, perform code generation, then throw the seed away. The only reason to keep the seed would be to consistently re-generate the whitebox code, should that be necessary for some reason.

Like secrets, a seed string should be generated with a strong random number generator and be at least 32 bytes in length, unencoded. See the previous Mac OS X command line as an example of generating a random string.

Once you have your seed string, here's how you provide it to wraptool (use your own seed, of course):

```
--seed "359260F0BEABEA0AA478F09C940ADEDDDC605D8A6F262D11836C757BDEC111E1"
```

Optional: Testing the generated Dynamic Crypto code:

By default the dcgen operation will also create, build, and execute test code that validates the generated code. If a license for one of the products in the wrap config is present during code generation, then the test code

will find it and perform encrypt and decrypt operations. These operations are timed, and the elapsed times are displayed to the developer.

Once the test is performed, wraptool will delete the test project, intermediaries, and the test application itself. If you wish to keep these test files instead, provide this option to wraptool:

```
--keeptest
```

If you don't want wraptool to test your generated code, you can provide this option:

```
--skiptest
```

Since the test code will not be built or run, the above option will save some time.

Optional: Advanced options:

Currently the dcgen operation uses source code templates as the basis for code generation. Included in these templates are the whitebox input and simulation sources. By the default the template files are copied from an installed location in the SDK. These copies are then subject to some string replacement, then fed through the whitebox generation process. Once the generation process is complete, the template copies are deleted.

For support reasons, and in case advanced developers are curious about the whitebox generation phase, the following option will prevent the deletion of the template copies:

```
--keepgen
```

Important! You should never include the templated code directly in your projects. Doing so would expose your secret keys to attackers.

Since the Dynamic Crypto code is provided in template form, it is possible to change that code. Reasons for doing this might be to make custom changes to the implementation known only to your company. If you make changes to the code that is fed into the whitebox generator, then any changes would also be highly obfuscated and unique.

That having been said, development of whitebox code is not for the fainthearted. There are numerous limitations that you have to adhere to, including not allocating memory, not calling external functions, and implementing all logic as deterministic code (i.e. no 'if' statements). Coupled with the fact that we currently have little documentation for the whitebox tools themselves, it would be challenging to successfully implement your changes, unless they are very limited.

But for those that wish to experiment, we suggest that you copy our template folder and make changes in your copy. Once done, you can use the following option to tell wraptool where to find your template files:

```
--template "MyCrazyTemplateChangesDir"
```

wraptool encryptcontent

This operation encrypts content using a content key file that was generated from the codegen operation. The encrypted content can later be decrypted using the PACE Eden interface in the presence of the auth allowed by the license data corresponding to the encrypted content. The decryption will only be possible if: (1) the license data that was generated along with the content key file is present and (2) an auth corresponding to a product defined in the license data is present. The products defined in the license data are the same products that were defined in the Wrap Config, which was used to generate the license data in the codegen operation. Be sure that your content key file and license data are a matched pair (both files generated together or the license data generated from the content key file using the codegen command); otherwise your decryption will fail.

You can encrypt your content directly, or you can encrypt a key that, when decrypted, can be used in your own decryption algorithm. Encrypting a key gives you more control in the decryption of your content, since you are using your decryption algorithm and code to do the actual decryption of the content. Encrypting your content directly is easier, but it may not exactly meet your needs. When encrypting data with this command, be aware that you will have to decrypt the data in exactly the same order. You cannot randomly choose a block at the end of the encrypted data to decrypt. You have to decrypt all of the data from the beginning to the block that you want to decrypt. So, if you have separate pieces of content that you need to decrypt separately, you should encrypt those pieces separately. For an overview of the content protection, please see the "PaceEdenContentProtection.pdf" document, which is included in the PACE Eden SDK.

Optional Common Arguments:

```
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
--verbose
```

Required: A path to a content key file generated by codegen:

```
--inkey MyExistingContentKeyFile.cke
```

Optional: A path to the unencrypted content file (a standard binary data file):

This is required to encrypt content. However, if omitted, then this operation will display information about the content key file.

```
--in unencryptedContentData
```

Optional: A path to save the encrypted content as either a binary data file or as a text file containing C code:
This is required if the inkey option was provided. If a file already exists at the given path, then this file will be replaced.

```
--out encryptedContentData
```

Optional: The number of blocks to skip after each block that is encrypted:
For content with a lot of data, skipping blocks of data in in the content encryption allows better performance, but does expose some of the unencrypted content. Zero means encrypt every block; one means encrypt every other block; two means skip two blocks after encrypting each block. A block is 16 bytes in the encryption algorithms used by PACE. It is up to the caller to keep track of the number of blocks to skip for each piece of content. Callers can create an algorithm that uses the content size to determine the number of blocks to skip. The publisher code that decrypts the content will be required to provide this number to the PACE Eden interface for each piece of content. Here are the options for providing the number of blocks to skip:

Provide this option specifying the number of blocks to skip.

```
--blocksskip 1
```

OR don't provide this option in order to use the default of zero (no encryption block skipping).

Optional: The type of encrypted content file:
You can have the encrypted data exported to a C code text file or a standard binary data file (the default). The advantages of the C code text file is that it contains comments about the encryption parameters and is more convenient to use with source control systems. The disadvantage of the C code text file is that the file sizes will be bigger than a standard binary data file. The encryption process does not increase the content size; however, the C code text file format inflates the size due to the comments and the text representation of the data. If you are concerned about data size, then use the standard binary data file format.

Provide this option to export the encrypted content as a text file containing C code:

```
--exportcode
```

OR don't provide this option in order to export the encrypted content as a standard binary data file.

Optional: A product specific key for the encryption:

Provide the auth ID that identifies the product in the content key file (from the Wrap Config). Any content encrypted with a product specific key can only be decrypted with the auth matching the auth ID provided here:

```
--authid 0x1234ABCD
```

OR don't provide this option in order to use the license data common key. Any auths matching the products defined in the license data can be used to generate the license data common key.

Optional: The publisherId of the product specific key:

If the authid does not uniquely identify a key in the content key file (a very, very rare situation), then you can provide the publisher ID. This is ignored if authid is not specified.

```
--publisherid 0x00000014
```

OR don't provide this option if authid does uniquely identify a key in the content key file.

wraptool signcontent

This operation is used to sign content. The operation normally needs a special publisher iLok2/iLok3 that has been updated with code/content signing credentials. Users can use the ILok License Manager App to get a code/content signing certificate, just by right clicking on the iLok icon and selecting "Synchronize" from the popup menu. Only users with the appropriate privileges will get a code/content signing certificate. If a user is associated with multiple publishers, then the certificate will contain the credentials to sign code/content for any of those publishers.

For performance reasons, using iLoks for signing code and content is best. However, in some build environments, such as building in the cloud, iLoks cannot be used. To address this problem, PACE has created an internet service that will do code signing in place of a signing iLok. This signing service is sometimes referred to as Cloud Signing or the Cloud Signing Service. This service should not be confused with iLok Cloud, which is PACE's internet-based licensing system. The --allowsigningservice option enables wraptool to use the Cloud Signing Service. ILoks, if connected to the computer running wraptool, are always favored for code signing; so be sure to unplug your iLoks if you want to try out the Cloud Signing Service. Also, for this option to work, your build environment requires an Internet connection. Additionally, you need to specify an account and an account password to a user that has the privileges to sign for the publisher specified in the wrapconfig.

Typically, content is signed to prevent the content from being tampered with and to prevent a content player from playing unauthorized content. Since encrypted content is ultimately decrypted on the end-user's computer, encryption cannot cryptographically ensure the above purposes. Signing, which uses asymmetric keys where the private key is never exposed on the end-user's computer, can cryptographically ensure the above purposes.

You can either sign before you encrypt your content or after. If you sign before you encrypt your content, then you have to decrypt your content before verifying the signature. This is actually the preferred method, because it verifies that the data decrypted correctly. However, if you have a reason to sign after you encrypt your content, then you must verify your signature before decrypting your content. For an overview of the content protection, please see the "PaceEdenContentProtection.pdf" document, which is included in the PACE Eden SDK.

Optional Common Arguments:

```
--account MyUserAccountName
--password MyDeveloperPassword
--binaries MyWrapperBinariesDirectoryPath
--installedbinaries
--allowsigningservice
--verbose
```

Required: A path to the unencrypted content file (a standard binary data file):

```
    --in unencryptedContentData
```

Required: A path to save the signature data as either a binary data file or as a text file containing C code:
If a file already exists at the given path, then this file will be replaced.

```
    --out ContentSignature
```

Optional: The security level of the signature:
Provide this option to set the security level of the signature. A security level of 0 generates a signature that uses a SHA1 hash and a ECC_Secp160r1 PK (Public Key) Standard. A security level of 1 generates a signature that uses a SHA224 hash and a ECC_Secp192r1 PK Standard. A security level of 2 generates a signature that uses a SHA224 hash and a ECC_Secp224r1 PK Standard. A security level of 3 generates a signature that uses a SHA256 hash and a ECC_Secp256r1 PK Standard.

```
    --security 3
```

OR don't provide this option in order to set the default security level of 0.

Optional: The publisherId of the credentials to use for the content signing:

Provide this option only if you are in the unusual situation of having signing credentials for multiple publishers. This ensures that the correct credentials are used to sign the content.

```
    --publisherid 0x00000014
```

OR don't provide this option in order to sign with the credentials found on any available iLok2 or iLok3.

Optional: A path to save the certificate chain as either a binary data file or as a text file containing C code:
This is optional, because only one copy of the certificate chain is needed for all the signed content. If a file already exists at the given path, then this file will be replaced.

```
    --certchain CertChain
```

Optional: Specify the type of signature data and certificate chain file:

Provide this option to generate the content signature and certificate chain as text files containing C code:

```
    --exportcode
```

OR don't provide this option in order to generate the content signature and certificate chain as standard binary data files.

wraptool repair

In the unlikely event of a WrapTool crash, access to the wrapper cache and other resources may be left in a locked state. Relaunching and attempting to access the cache and these resources may result in a hang due to the lock. The repair operation releases the lock, so that WrapTool can function normally again.

```
   Optional Common Arguments:
          --verbose

   Examples:
      wraptool repair
      wraptool repair --verbose
```