Apollo

License verification specification

Confidential

Revision number: 1

Date:

Name: P. van der Velde

# Disclaimer

This specification is not, by any stretch of the imagination, complete. It will need to be revised several times before it is complete. Currently several major parts are either missing or incomplete. This disclaimer will be updated to reflect any change in these sections. Finally a specification is supposed to be a ‘living’ document and therefore never complete. What were you thinking, better learn to live with this fact.

# Introduction

The whole goal of the licensing components is to keep the honest user honest. In other words the license components are there to ensure that the user complies with the license terms, i.e. only the number of installs they paid for, no sharing of licenses with others, etc.. The implementation of the components is not setup to deter professional crackers, although some of the implementation is aimed at deterring the casual cracker.

In the design and implementation of the license components the main mantra will be:

Do not annoy the paying customers!

The actual generation and verification of the license keys will be done through a third party API because the implementation of key verification is complicated. The Apollo license components will wrap the third party API and handle the invocation of the third party API and the response to any verification errors.

Finally it is important to note that the verification components are only a small part of the license protection system. Other methods belong are provided through the software distribution methods. For more information on these methods please review the distribution document.

# Licensing code

This section discusses the demands placed on the third party licensing software that will be used for key generation and validation.

The licensing software needs to be able to:

* Easily to generate license keys. Each generated key needs to be unique and bound to a specific user. The binding may be through the user name or machine identifiers.
* Each license key should be bound to a specific version of Apollo. It should not be possible to use a license key for a version of Apollo for which the key was not generated.
* Generate temporary licenses for beta or trial versions. Also temporary licenses can be used for software leasing where the customer 'rents' the software or extra modules for a specific time.
* Handle network floating licenses that allow a customer to have multiple licenses that can be used from multiple machines.
* Handle black-listing of known-compromised keys.

The third party software does not need to be able to handle 'call-home' verification because there is no guarantee that the customers computer will be able to connect to our own servers via the internet.

Finally it is not required that the licensing software comes with obfuscation tools because those tool will be obtained from a third party (not necessarily a different one). The obfuscator can then also be used to merge the licensing code into the main assembly of Apollo.

# Implementation

The license components for Apollo handle calling the third party license API and

* Create a verification class which receives the results of a verification. Each service can hold on to one of these. Verifications time out after XX time. When no valid verification is found then the service should stop functioning and send a request for new verification. If no verification is available within YY time then we stop and notify the user
* The verification class can also handle the requests for the encrypted code
* Which kind of key validation will we use? Partial verification sounds like a good plan
* How will we deal with an external license service?

Another thing to think about is the security messages. They should never be displayed directly. This is where the messaging system comes into play. First we have to send all messages for the user to the UI via the message service. We could include a key in the messages which indicates where to get the actual translated message (from a resource file). We’ll need to see how to do this though because we don’t want the UI to load all the assemblies. Only system assemblies should be loaded. So in that case we’ll have to send a message back into the AppDomain where the message came from and load the data there.

Suggestion: can have a licensecheck struct / object which has a checksum. External code then checks the checksum (spreads the booleans around)

Steps to take:

* When a license verification fails then we should (eventually) send a message to the user. The best thing to do is to send a generic error telling the user to contact support. Provide a set of error codes (say 10 different ones).
* Have multiple verification routines (code generation can help here).
* Have multiple verification layers. Run these layers on different threads (but only a few different threads. One thread can run multiple layers). Ensure that the verification threads set a expiring flag that indicates that they are alive.

How does the checksum bit work?

## Time based verification

* Run one or more thread based timers. Each timer has a random time interval with a small, medium or large time base (e.g. minutes, hours, days, weeks, months). When the timer fires we validate the license. Time can be based on last start-up, install date or other.
  + A hourly timer for the current run
  + A daily timer based on the last start-up set (maybe the last one in a set of 10, or a random set with a maximum size of 50). Maybe also include the last check date
  + A weekly timer based on the last start-up set (ditto to the daily one). Maybe even include the install date or the last check date
  + A monthly timer which is based on the install date + last check date
  + A random time timer which is based on the install date + last check date.
* The threads must be made bullet-proof so that injecting thread errors does not lead to a failure of the license system. A dying thread should lead to creating a new one and starting that.

## Encrypted code store

* Encrypt code during compilation. Only provide store with a public key which is based on the license key

The obvious questions are:

* How do we hide code in an encrypted block
* Which pieces of code will we hide
* How to determine that code can be decrypted / when etc.
* How do we make this safe so that the user can use it but not crack it ... mmmm this seems to be a paradox.