Axantum Strong Software Licensing

Svante Seleborg

Axantum Software AB
svante@axonadata.se

Abstract

This specifies the functionality and technical details of the Axantum Strong Software Licensing feature. The technical background and usage of respective programs are described.

The licensing software is implemented in the AxCrypt file encryption software, available as an open source application as well as licensed by various OEM vendors.

Although AxCrypt is free and will remain so, both itself and the software licensing part described here, is available under commercial terms from Axantum Software, and AxCrypt with this software licensing scheme is intended to also serve as a strong technology demonstration for interested OEM parties.

Axantum Strong Software Licensing offers two relatively unusual features, still hard to find implemented in free or commercial software today, in 2004:

- Self-signed code and configuration using strong elliptic curve cryptology conformant to international standards.
- Compact short signature technology using relatively strong elliptic curve cryptology enabling keyboard entry of a true digital signature. This is one of the first available implementations of shortened ECDSA signatures based on published research and built on top of international standards.

1. Introduction

Software Licensing is a variation of Digital Rights Management, i.e. the art of retaining control of works distributed digitally – the challenge being that faithful copies of digital works are trivially made and distributed.

Retaining control over distributed works in the digital world is in this context equal to restricting the possibilities of unlimited copying, usage and distribution in one way or another.

AxCrypt contains a software licensing and digital signature verification feature, which has the following design goals:

- Copying as such is not restricted, i.e. the software is not tied to a single machine or media – although that may be implemented as well within the scheme.
- The risk of widespread copying, distribution and the therefore unlicensed usage is reduced by requiring a digitally signed license-to-use, which positively identifies the licensee. Thus – if copies are distributed, the source may be traced.
- The solution is cryptographically strong; it is not an obfuscating scheme.
- It allows the same binary compiled program executable files to be used both in a licensed and a non-licensed mode.
- As a benefit to the user, the scheme also provides strong digital signatures over essential program files, ensuring self-validation resulting in safe and uncontaminated execution.
- "Non-validated", externally signed, XML is used for configuration.
- While circumvention can always be achieved by skilful patching of the executable, this will by definition also lead to loosing the program validation feature, thus a patch does not retain full program functionality.
- The license-to-use is a relatively strong digital signature typically over a user-identifying string issued such as an e-mail address, issued by the software manufacturer.
- Forging of program-code and configuration validation signatures is not computationally feasible.
- Forging of license-to-use signatures is computationally hard, though possibly feasible in the future as Moore's law does its work.
- The license-to-use signature is short and compact enough so it's reasonable to manually enter with the keyboard, although cut-and-paste is recommended.

2. How does it work?

The software consists of a number of executable files, specified in the configuration XML-file, for AxCrypt, typically AxCrypt.exe, AxCrypt.dll, AxCryptM.dll and Notify.exe.

The configuration XML-file, typically named Config.xml, contains all static program configuration information that is not stored in the registry. This includes signatures for the component programs, visible names etc. This is never modified after distribution, as it is digitally signed in an external file, the Signature XML.

The Signature XML-file, typically named Sigs.xml contains signatures and other dynamic information. This file is not signed.

AxCrypt is hard-coded to find the Signature XML in the same directory as the executable, and is typically named Sigs.xml. The name is hard-coded in the executable. If it does not find it, startup fails. Sigs.xml refers to the configuration XML-file. If it is not found, startup fails.

2.1. Signature XML - Sigs.xml

Sigs.xml typically contains:

```
<Signatures>
  <Config>
    <Signature File="Config.xml" >
      HexString
    </Signature>
  </Config>
  <Licenses>
    <Signature Terms="Full"
Licensee="User Name etc">
      Base34String
    </Signature>
  </Licenses>
</Signatures>
```

Upon startup, AxCrypt attempts to locate the Signature XML file, by way of a hard-coded name like Sigs.xml, in the same directory holding the executable. The main purpose of Sigs.xml is to hold the signature of the Configuration XML. This makes for easy and unambiguous signing and verification. The Signature XML may also hold system-wide blanket license signatures issued with the distribution.

User-entered, volatile licenses may be stored in any medium appropriate for the environment, typically the registry for Windows – or if appropriate by modifying the Signature XML, this requires upgrade procedures to merge XML though.

Startup proceeds by parsing the Signature XML, and locating the <Config> tag. It starts by checking the specified signature against the file found. If it does not match, start up fails. The public key is hard-coded inside the AxCrypt executable. The private key is the property of Axantum.

As a consequence only Axantum Software can produce valid Configuration XML-files; vendors or other parties cannot unless a patch or a recompile inserts a different public key inside the executable.

Program upgrade is performed by replacing the appropriate executables, Signature XML and the Configuration XML

If several license signatures are found, all are matched against each entry in the list of license types in the Configuration XML.

2.2. Configuration XML – Config.xml

Config.xml does not change during the life of a version installation and typically contains:

```
<Configuration>
  <Self File="AxCrypt.exe" />
  <Signature File="AxCrypt.exe">
    HexString
  </Signature>
  <Signature File="AxCrypt.dll">
    HexString
  </Signature>
  <Signature File="AxCryptM.dll">
    HexString
  </Signature>
  <ShortName>
    AxCrypt
  </ShortName>
  <LongName>
    Axantum Encryption
  </LongName>
  <RegistryPath>
    Axantum Software\AxCrypt
  </RegistryPath>
  <Restrictions Days="20" Uses="25">
    AxCrypt
    <Verifier>
      HexString
    </Verifier>
    <Terms Days="" />
    <Terms Uses="" >
      Small
    </Terms>
    <Terms Days="" Uses="" >
      Full
    </Terms>
  </Restrictions>
</Configuration>
```

As Config.xml is signed upon distribution it can never change, and must only contains things which are static from installation and onwards. If upgraded, it must be matched with a corresponding Signature XML – thus as long as the embedded public key remains inviolate the configuration can not be modified for a given executable. This self-testing introduces an exceptional resiliency against manipulation and programming errors.

If there is no <Restrictions> tag, no restrictions apply. The value of the <Restrictions> element identifies the restriction group.

The internal working of license restrictions are in essence beyond the scope of this document, it's up to the main program to enforce the restrictions until a

valid license-to-user is presented and entered into the Signature XML.

The attributes of the <Restrictions> element list the default restrictions that are in place.

The list of <Terms> is interpreted in order, as are the attributes. Each <Terms> element that does not have a valid license is ignored. If it has a valid license, it's attributes are applied. Empty attributes remove the restriction. Only restrictions are listed – never rights. The default is thus all rights, contrary and in reverse to many other rights systems. If a <Terms> element does not specify a given restriction attribute, that <Terms> element never has any effect on that restriction.

It is up to the implementation to recognize and interpret various restriction attributes. Non-recognized restrictions are simply ignored. Typically for AxCrypt there will be support for a counter and possibly a day counter from installation after which further encryptions is not allowed, but apart from that the program continues to work.

The free version, which of course should have no restrictions, is distributed with a license signature in the Signature XML granting full rights, and a <Restrictions> section in the Configuration XML. This is only useful for the free version, since the Configuration XML requires that the executable be named AxCrypt.exe, and that the registry settings are stored in the place for the free version.

To enable independent developers to compile and run AxCrypt a Configuration XML / Signature XML pair is distributed which does not require any signed files at all. When using this Configuration, a warning dialog is shown which advises the developer to turn the whole signature feature off for independent development, since the signing tools are not distributed. This is also used for debugging purposes.

Circumvention requires patching the program executable to either skip the tests for validity or insert a new public key and then sign everything with the corresponding private key.

2.3. What is required to break the system?

Essentially patching the program is required to break the system from an executable point of view. Obviously, it is always possible to simply recompile the source since it is open for view and download, but the point is that it is not feasible to install a protected version and then patch it without loosing the self-validation feature. Possible, yes, but it requires a bit of work – work that requires significantly more effort than simply down-loading the free version.

It is not computationally feasible to forge the configuration XML. The program has a hard-coded public key and it requires finding and validating the configuration XML before starting.

It is at least computationally very hard to forge a license-to-use, probably it is also infeasible at the time of writing and for years to come.

Obviously, if the same binary is used in different contexts – the free version or varying vendors OEM-versions, it's possible to replace the configuration XML and signature XML with data obtained from the free version. But since the executable must also be renamed to the free version, the registry data moved etc – what is thus achieved is an installation of the free version in an extremely round-about way. That is no problem, as it's no secret there is a free version – but that comes without any support, any distribution etc. For a fully commercial software, it's no limitation at all since there will be no free signed configuration files available.

2.4. The licensing process as viewed by a user

A user purchases the program, and gets an installable on some media, including online download.

Along with this, she receives a *proof of purchase*, or *serial number*, or any other token that may be presented at the vendors web site or via e-mail, phone, fax or phone or even in direct contact with the vendor. **This mechanism is not part of the AxCrypt Software Licensing.**

The vendor, by any mechanism of her choice, then receives a string representing the user identity. Typically this will be the users e-mail, real name or similar. **Validating this identity is not part of the licensing scheme.**

The user, in return, receives a communication in any form consisting of the *identity string* and a *license*. The identity string may look like this (only characters and letters are significant, and the case is not):

```
svante.seleborg@axondata.se
SVANTESELEBORGAXONDATASE
```

Both above are identical. The purpose is to be flexible if the string is typed manually.

The license takes the form of 36 significant characters, optionally grouped for easier legibility, and may look like this:

```
IE5EPX-5IEDU3-6XJWYB-AEB3E5-MZIC7A-N4FCC7
```

The possible characters include A-Z except 'O' and 1-9, for a total of 34 different characters and the case is not significant. Note that letter 'O' and digit zero are excluded to avoid confusion.

The user, upon receiving these to strings, is prompted in the same communication to launch the license wizard, which is a simple program where the two strings are entered in two respective edit boxes, after which the user presses 'OK'. If they are valid and correctly entered, the program now runs without any restrictions. If not, the user is prompted to try again.

2.5. What are the vendor tools?

The vendor receives one command line tool in a Windows and a FreeBSD/Linux version, along with a private key generated by Axantum Software. The private key generation program is not made available to external vendors, but private keys are generated upon request according to agreement (priv.key below, a small text-file of a few hundred bytes).

The tool, AxSigLic (or axsiglic on FreeBSD/Linux) takes the following arguments:

```
axsiglic –r priv.key [-t type] –m "user-id-string"
```

The resulting license/signature in the above-mentioned 36-significant character format is produced on stdout, redirectable at will. The type/label is prepended to the licensee user-id-string and after canonicalization used as the message to sign. The canonicalization removes all non-alphanumeric characters and converts all to lowercase ASCII/Ansi 8-bit codes.

The private key file is a raw hex encoding, as produced by AxKeyGen.

The tool may be incorporated into scripts, web-pages, GUI-programs etc according to whatever the vendors need is.

2.6. What are the private tools?

To explain and document the full workings a description of the Axantum private tools follows. These are not distributed in either source or binary form to discourage generation of 'false' private keys. Distribution and generation of 'false' private keys is not a security issue as such, since it still requires recompilation for modification of the binaries to be of use.

2.6.1. axkeygen – Key Pair Generator

Private keys are generated very seldom; Axantum has one and will under normal circumstance never require another one. The same applies for vendors and technology OEM's – one is generally enough. Thus the usage frequency alone makes the non-distribution of the tools reasonable. The tools are not documented or packaged in release quality either. They are for Axantum internal use only.

The private key generator can generate two types of key-pairs:

- License-to-use key-pairs. The private key is used to sign user identifications to produce the short license-to-use signatures consisting of 36 characters as explained elsewhere.
  The public key will normally be distributed in signed XML to be used by applications to validate license-to-use signatures.
- Code validation key-pairs. The private key is used to sign code, configuration files and any other data where the signature is never typed by a human, but carried by higher bandwidth means such as files or the internet.
  The public key is typically either compiled statically into code, or placed in signed XML.

It is called on the command line as follows.

```
axkeygen [-l | -s] -[c|h]u "pub.ext" -[c|h]r "prv.ext"
```

-l long signature validation key-pairs.
-s short signature license key-pairs.
-c output is a C code fragment
- h output is raw hex
-u file name for the public key
-r file name for the private key

Default output is XML.

2.6.2. axsigxml – Configuration signer

Using the private validation key, a XML file is parsed and the appropriate signatures for referenced files are generated etc. It takes as input an existing Configuration XML file, and produces as output a version with the appropriate signatures.

Usage:

```
axsigxml [-r prv.xml] [-p searchpath] [-t tag:file]
```

-r private key to sign with. Made with axkeygen.
-p folder to search for files in. May repeat.
-t replace element with contents of file.

It reads stdin, and produces to stdout. Elements named <Signature> with an attribute File are used to find files, generated signatures, and then output them as element data.

2.7. Technical details

2.7.1. Rights Manager

Having established signed and secured configuration data and binary executable integrity, there's a need to actually manage the rights conferred by the license. Generally speaking it's about limiting the functionality in run-time conditional on both static and dynamic checks.

Static tests refer to simply limited license, for example simply restricting functionality. This is a limitation that is is statically tied to the license given – a new, better, license needs to be issued for the functionality to be available.

Dynamic tests refer typically to time or number of uses restricted functionality. I.e. after 30 days, limit the functionality or after 25 uses. These differ from the static kind in that they need to dynamically store and update data in a reasonably secure store that is at least not trivial to manipulate. For example, having a use counter in the registry falls to a trivial resetting attack.

The challenge is that "by definition" nothing offline is truly secure, except, but one axiom in this whole context is that we regard the executable file as inviolatable as it is signed. If this is bypassed, there is nothing to stop simply modifying the executable to skip the limiting tests, or simulate a full valid license.

The AxCrypt Rights Manager is a program API which takes as input an internal representation of an XML tree representing the Signature XML and one representing the Configuration XML. The license conditions are expressed as a number of restrictions that apply if the license is not valid.

To store the state of use-counts etc, the following strategy is used. It is a low-bandwidth strategy, only a few bytes of information may be stored in this manner, but it should suffice. The main purpose is to ensure there are no pre-installed tools to easily delete the info.

A key container is created with a pattern that is recognizeable, i.e. begins with AxCrypt for example. Directly following this, hexencoded, follows the use-state structure as defined by the implementation.

When no matching key container is found, it is assumed that it's a new install and default zero counters are created.

The state is updated by creating a new key container and deleting the old.

Uninstall removes all information, except the key-container it-self. This will let a subsequent reinstall continue from where it previous installation left off.

To defeat, someone must write a program that enumerates the key containers and delete the one used here, whereupon the program of course reverts to new installation state at the next startup. If this becomes a problem, an arms race can be started where a key-pair is generated and stored there too, and used for various purposes to stop simple deletion from resetting the state.

2.7.2. Signature technology

(To be completed after implementation is complete)

Elliptic curves over GF(p) are used. For the internal validation of the configuration file, one of the recommended 384-bit curves is used with a standard ECDSA signature algorithm.

The signature over the user identication which produces the license-to-use, is done with one of the recommended 128-bit curves. The signature process is modified according to principles outlined in the references, by letting the parameter 'r' in the signature be a truncated hash – here set to 55 bits, making a total of 183 bits which is what fits into 36 base 34 positions. The choice of parameters here may change, and there's a perception that it is possible to add a work-factor increasing method to the hashing so as to increase the effective security offered by the admittedly short hash of 55 bits. Please note that in this application the birthday paradox weakness should not apply – a brute force attempt to force a signature requires finding a message that hashes to the same hash as a given message, not just to find two which collide. A different possible tradeoff is to use a 112-bit curve with up to 71 bits of the hash. The current choice is made to protect the private key as well as possible. The weak hash should not affect the strength of the curve as such, only the ability to generate a single false signature. It is not entirely clear exactly what the complexity is to force this combination of parameters. Normally the scheme is regarded to give equivalent complexity as the regular scheme over the same curve with a hash of half the size of the curve, in this case 64 bits. 128-bit curves are also small, but currently beyond demonstrated abilities to force. In 2002 a 109-bit curve challenge was solved by utilizing a massive amount of computing power including 10,000 computers (mostly PCs) running 24 hours a day for 549 days.

2.8. Intellectual Property Rights

A reasonable amount of effort has been spent to determine if this implementation or its techniques are covered by any patents or other IPR. No such evidence

has been found. The techniques around ‘Signcryption’ are probably protected, but that is not what this application is about. The reference is only used for the shortening of the ECDSA signature. All algorithmic code is from the Crypto++ library which is in the public domain, and no warnings about IPR are evident concerning this usage.

3. References

[1] Zheng, Y., ``Signcryption and Its Applications in Efficient Public Key Solutions"'.

[2] Zheng, Y., “Digital Signcryption or How to Achieve Cost(Signature & Encryption) << Cost(Signature) + Cost (Encryption).

[3] Zheng, Y., Imai, H., “How to construct efficient signcryption schemes on elliptic curves”, Information Processing Letters 68 (1998) 227-233.

[4] Johnson, D., Menezes, A., Vanstone, S., “The Elliptic Curve Digital Signature Algorithm (ECDSA)”, Certicom Research, Canada.

[5] Menezes, A., Oorschot, P. van, Vanstone, S., “Handbook of Applied Cryptography”, CRC Press, 1996.

[6] Dai, W., “Crypto++ 5.2.1”, http://www.eskimo.com/ ~weidai/cryptlib.html.