Policy in Public Key Infrastructures, or How Values End Up in the Code

Brian A. LaMacchia[1]
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`bal@microsoft.com`

Abstract

Whatever societal and ethical values we wish to include in a computer system must eventually make their way into the software itself and become part of the system's code. Although authors of computer systems actively attempt to make their products "extensible" and value-neutral, in fact every software design decision made as part of the process of building a system is a reflection of a value system. Application programming interfaces (APIs) and service provider models, used generally to build flexibility and customizability into software, also allow applications to override the biases built into the code and thus construct alternative value systems. Even when a system is explicitly designed to be value- or policy-neutral, such as the current proposal before the IETF for an X.509-based public key infrastructure, the extensibility of the infrastructure exists as a result of the underlying value systems of its creators. Ultimately, software developers must acknowledge that it is not possible to create a value-neutral system simply by adding extensibility to a biased system.

1 Introduction

Whatever societal and ethical values we may wish to include in the development and deployment of a computer system, if those values are going to impact the system then at some point they must be incorporated into the software itself. Policy options (defined by value systems) are incorporated into a product like any other functional requirement or feature request: we create interfaces and share code where commonality exists, and we provide platforms that allow customization where customer-specific solutions are required. The requirements and requests themselves come from our customers, be they individuals, enterprises, governments or society as a whole. After the system is deployed, customer

[1]The opinions expressed in this paper are those of the author and are not necessarily those of Microsoft Corporation.

feedback helps shape the next generation of the system and (hopefully) create a better product.

Software vendors primarily create extensibility in systems using application programming interfaces (APIs) and "plug-able" service provider models. Recognizing how APIs and service provider models are incorporated into current software is key to understanding how policy and value systems end up being reflected in source code. An API is an interface in the code that is, in essence, a contract between the "lower-level" system and "higher-level" applications[2]. When the system publishes an API it is claiming that some core functionality is sufficiently common that multiple applications will wish to use it. That functionality is then provided by the system itself; application authors build on top of the API.

At first glance, the establishment of an API appears to be a value-neutral act. However, creating an API is not only a statement made by the system developers (that they believe commonality exists at the interface boundary) but also a value judgement. An API is, in fact, a belief claim that *all* applications wishing to use the functionality provided in the API wish to do so in the manner provided by the API. Depending on how the API is constructed and exposed, application developers may be forced to comply with a policy embedded within the API in order to access a lower-level resource. Often this is desired, e.g. access to a disk drive is serialized via policy embedded in filesystem APIs. (Imagine the chaos that would ensue if every application wrote to the disk whenever it wanted to without coordination by the underlying operating system!)

Sometimes, however, the API is an expression of belief in only the interface and not in the underlying mechanism providing functionality. Multiple implementations of the API are possible, each perhaps with strengths, weaknesses and side effects. In these cases an API is defined to include not only the interface but also the ability to install, use and choose among multiple *service providers* of the underlying functionality[3]. Typically at least one such provider is included with the API as a "default" or "base" provider, but use of the default is not mandated. Developers may, if they choose, create their own alternative providers and install them; the only requirement on an external provider is that it faithfully implements the API. Applications calling API functions specify which provider they want to use (the context of the call) in addition to the nominal arguments to the function.

The main benefit of the API/provider model is that it allows the publisher of the APIs to abstract and create layers of common functionality. A side effect

[2]Here the terms "system" and "applications" are used loosely to correspond to two layers of code: code that implements the API (system) and code that uses the API as a pre-existing component (applications).

[3]Open Services Architecture [4, 5], for example, is an example service provider model. WOSA provides a set of APIs for accessing classes of services (like database servers). Client applications that wish to use a database service call the WOSA database APIs, and database implementers provide the "plug-ins" that translate WOSA database calls into vendor-specific APIs.

of the proliferation of APIs and provider models, though, is that developers are able to avoid making traditional "policy" decisions in the code through their judicious[4] use. To illustrate how we create "hooks" in the code for whatever policy and value systems are desired we need only look at current efforts to construct public key infrastructures (PKIs) for electronic commerce. Through the use of APIs, provider models, extensible data structures and arbitrary indirect references, the PKI community has succeeded in specifying an infrastructure with a totally flexible unspecified value system. Interoperability has been preserved at the data structure level, but the semantic meaning of elements must be agreed upon out-of-band by PKI applications.

2 Building PKIs

Creating a digital signature on a document is easy; methods for doing so with public-key cryptography have been published since the late 1970s. Validating a digital signature is also a simple task, assuming that the validation engine possesses the public key corresponding to the private key that created the signature in the first place. The difficult problems involve interpreting a digital signature once the mathematics has been verified. For example, how should applications decide whether to trust a particular digital signature and what are the consequences of that trust policy? More generally, a digital signature is simply evidence supplied to a policy evaluation engine [1, 2], so somehow that signature must acquire a semantic meaning.

In the physical world signed statements acquire meaning through agreed-upon societal structures and conventions. Both positive and negative reinforcement may be used to build trust. The accumulation of credit or reputation capital in an identity serves as a deterrent to socially unacceptable behavior; "bad acts" cause a loss of reputation. When there is no reputation capital at risk the force of law (that is, fear of civil or criminal penalties) is used as the basis for trust. The digital domain requires similar conventions among its participants; PKIs attempt to provide them. PKIs are sets of relationships among digital principals; participants in a PKI make statements about other participants and together those aggregate statements provide a framework for reputation. (The most common statements in today's PKIs are certifications of identities and bindings between identities and public key components.) This collection of statements, embodied in the PKI, in turn drives policy evaluation when digital signatures are used in a decision-making process.

Both the semantics of a particular digital signature as well as the process of choosing whether to trust that piece of information depend on value choices. These value choices may conceivably vary from community to community, and the "community" could be a single individual, an enterprise, or the entire Internet. Thus, we find commonality in specification but not in actual function and

[4]Some might say "liberal."

therefore we abstract the functional requirements via APIs and provider models. For example, consider the PKIX Part 1 draft standard for X.509-based public key infrastructures [3, 6], which is currently under discussion within the Internet Engineering Task Force (IETF). PKIX Part 1 specifies the format of a digital certificate, a binding between a public key and identity information concerning an entity with access to the corresponding private key. The certificate format includes mandatory and optional components as well as a private extension mechanism that allows parties to include their own custom data. Applications conforming to the PKIX standard are required to parse and understand the mandatory and optional fields within the certificate but only parse private extensions. Thus, PKIX-compliant applications will always be able to understand the syntax of the certificate but not necessarily its semantic content.

Within an X.509 certificate the semantics of the signature are specified via both machine-readable ("key usage" and "extended key usage") and human-readable ("policy qualifier") components, each of which may be extended as necessary using the private extension mechanism. Common uses of these two components are included in the X.509 standard (again, to guarantee a level of interoperability) but the issuer of a certificate (the "speaker" of the digitally-signed statement) always has the option of using his own custom variants. Thus we have results like this: every X.509 client should know that "1.3.6.1.5.5.7.3.2[5]" means "client authentication" (because it is part of the standard) but only certain browser applications will know that "1.3.6.1.4.1.311.10.3.3" means "allow Microsoft server-gated cryptography[6]" because that is a private-use extension. Obviously, the inclusion of particular semantic meanings in the standard reflects a value choice by its authors as to what uses are sufficiently common that support for them should be widespread. By creating extensible data structures in the standard we avoid having to make any value judgements concerning whether these certificates should be used for any particular purpose.

Where policy and value systems really impact the PKI, however, is when a "relying third party" attempts to use X.509 certificates to make a trust decision. The decision whether to accept a particular digital signature as evidence of a request or action must be made based on risk, trust relationships and societal factors at the receiving end of the transaction. The requirements may be simple: for example, the Internal Revenue Service might only accept identity certificates signed by the U.S. Postal Service. More complicated evaluation rules are certainly possible; "identity" may in fact be established through a reputa-

[5]This dotted string is an "object identifier" (OID), a unique string used as an reference in the X.509 standard. An OID is like a URL in that it refers to a unique location in a namespace, but OIDs cannot be de-referenced like URLs can.

[6]"Server-gated cryptography" (SGC) is a technology, introduced by Microsoft and Netscape in compliance with U.S. export regulations, that allows foreign banks to use strong cryptography in SSL/TLS connections between their Web servers and individual browsers. Servers that meet export requirements for strong cryptography are given certificates signed into a special rooted hierarchy; exportable browsers are able to negotiate 128-bit connections with servers that have such certificates.

tion network built solely out of financial instruments and the transfer of risk. Faced with these possible choices (all plausible) implementers must necessarily abstract away any particular evaluation method and create an API/provider interface for trust evaluation. Systems may then ship with default evaluation engines without forcing users to accept any particular model of trust validation.

Finally, we need to examine how actual implementations of a PKIX-compliant PKI modify the implicit value systems within PKIX. Extensibility is great for those building on top of a PKI but plays havoc with designers trying to build user interfaces (UI) for PKI objects. As implementers, we make our own judgement calls on issues such as:

- What features of the PKI are most likely to be useful to the majority of our customers (and therefore our highest priority for implementation)? Features not broadly useful to our customers are less likely to be implemented by us but left to third party developers.
- How do we represent both mathematical and semantic concepts involving digital signatures to users? For example, which errors or "defects" in a certificate hierarchy should be explicitly called out to the user for attention? How do we properly convey the seriousness of designating a new root certificate a "trusted root"?
- What information is presented to the user during the course of his interaction with the PKI? Even something as simple as name presentation is an important choice, as individuals will format names within their certificates so they display properly on our UI.

Here values end up in the code not though any conscious effort to embed a value system but simply as a result of the implementation process. When resources are limited[7] features are triaged and only the strongest (most demanded) necessarily survive. Usability is a major driving factor; grandma should not be forced to type in the "code signing" OID (1.3.6.1.5.5.7.3.3) in order to tell the system to check digital signatures on downloaded objects. At the same time, enterprise customers want the ability to completely customize their environment and are willing to write code as necessary to do things like introduce custom certificate extensions. We are constantly faced with usability vs. flexibility tradeoffs.

Most important to the value system the customer gets "out of the box," though, is the set of default options and functionality we provide for use with the PKI. Naïve users will not customize their environment (they won't even visit the option pages on Internet Explorer) so we must provide reasonable default behavior at installation without any user input. Default choices are everywhere, from components of the UI to the base providers included as part

[7]First rule of computer system implementation: resources are always limited.

of our API/provider models. The signature validation engine we ship as the default will be used by lots of users (perhaps almost everyone), so there is good reason to choose wisely.

In summary, we see that authors of computer systems actively attempt to make their products “extensible” and value-neutral to counter biases that are inherent in their creations. Although we may not like to admit this fact, every design decision made as part of building a system is a reflection of a value system. APIs and provider models are used to construct open platforms, which as a side effect serves to counter the built-in biases in the code and allow the insertion of components that construct alternative value systems. Of course, should the values of our customers change that same interface allows rapid deployment of updated system elements implementing those changes.

References

[1] M. Blaze, J. Feigenbaum, and J. Lacy, “Decentralized Trust Management,” in Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp. 164-173.

[2] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. “REFEREE: Trust Management for Web Applications,” World Wide Web Journal **2** (1997), pp. 127-139. (Reprinted from Proceedings of the Sixth International World Wide Web Conference, World Wide Web Consortium, Cambridge, 1997, pp. 227-238.)

[3] PKIX Working Group, Internet Engineering Task Force. “Internet Public Key Infrastructure: X.509 Certificate and CRL Profile,” work in progress. (Draft as of 10/14/1997 available from ftp://ietf.org/internet-drafts/draft-ietf-pkix-ipki-part1-06.txt.)

[4] Microsoft Corporation. “Windows Open Services Architecture (WOSA),” Microsoft Backgrounder, March 1992.

[5] Microsoft Corporation. “Windows Open Services Architecture (WOSA): Delivering Enterprise Services to the Windows-Based Desktop,” Corporate Backgrounder, July 1993, Part No. 098-53420.

[6] ISO/IEC JTC1/SC 21, Draft Amendments DAM 4 to ISO/IEC 9594-2, DAM 2 to ISO/IEC 9594-6, DAM 1 to ISO/IEC 9594-7, and DAM 1 to ISO/IEC 9594-8 on Certificate Extensions, 1 December, 1996.