TrustManagement

"Trust Management" is a term for how Shibboleth uses cryptography to secure SAML messages and assertions at a level beyond just determining whether the XML has been modified or not. Trust is an overloaded word, but here we mean simply the binary choice that the software has to make: does this bag of bits get used or rejected with an error?

SAML itself defines little or nothing related to this question. Mechanisms like SSL/TLS and XML Signatures, along with others, are spelled out as viable (and in some cases required) approaches to securing SAML profiles, but there are no guidelines or rules for how to make them usable and expose the configuration of policies and key management to the deployer.

As a result, the Shibboleth Project was forced to develop strategies for this purpose to implement in the software. These features give you as a deployer the necessary tools to manage your systems safely, but **you** are responsible for understanding them and using them properly. Failure to do so will result in serious exposures and open you to various kinds of external threats.

First Principles

In a lot of security software, including many SAML implementations, "trust" is implemented exclusively via a reliance on an X.509-based PKI, and the rest of the configuration is designed around this assumption. Much of the work involves telling the software about "trusted certificate issuers" or "certificate subject names". Some support for self-signed certificates is often present, but it can be difficult to set up or limited in scope.

Shibboleth takes a different approach, in that we base trust management on one or more plug-ins called Trust Engines. These plug-ins are able to implement any strategy desired for verifying signatures or TLS certificates without requiring modifications to the rest of the system.

The plug-ins that are supplied with Shibboleth are based on the use of Metadata to supply rules that the TrustEngine enforces to determine whether a particular key is "trusted" to be used by a particular IdP or SP. At runtime, an IdP or SP is configured with a set of metadata sources to use, and once those sources supply valid metadata to the running system, it is implicitly accepted by the rest of the software. In this manner, trust management is "bootstrapped" using Metadata. The ultimate security of the deployment really depends on both the accuracy of that Metadata and the way in which it's verified before being used. This is discussed in more detail below.

Inline / Explicit Key Trust Engine

The strategy recommended for use with all Shibboleth deployments is based on an "inline", or "explicit key", model and has been standardized at OASIS as the basis of the Metadata Interoperability Profile.

The technical details of this plug-in's behavior are laid out in the ExplicitKeyTrustEngine topic, and the relevant configuration references can be found here (IdP) and here (SP).

The idea behind this model is that the metadata for an IdP or SP explicitly identifies the public keys that it is authorized to use using `<KeyDescriptor>` elements. Certificates are used in most cases as a convenient "structure" to express the public key(s), but the key is all that matters, and deployers get maximum reliability (fewer obscure X.509-related errors) and flexibility. In particular, it allows a certificate in the metadata containing a key to be different than the key installed on a web site or inside the software, as long as the keypair matches.

It also has the nice property that **all** of the information in the metadata that has security implications is effectively bundled together. If you secure part of it, you secure all of it. In effect each `<EntityDescriptor>` becomes like a big self-contained certificate, which makes things a bit easier to grasp and to manage.

PKIX Trust Engines

A legacy strategy that we find creates a lot of difficulties for deployers and is not recommended in most cases is based on the use of "certificate path validation" at runtime. The technical details of this approach are laid out in the PKIXTrustEngine topic, and the relevant configuration references can be found here (IdP) and here (SP). There are versions of this plug-in designed around both "dynamic" and "static" approaches to providing PKIX trust anchor information (see below).

In addition, you must "glue" the SAML and X.509 worlds together by identifying the names of the certificates to allow a given entity to use by expressing them as `<ds:KeyInfo>`/`<ds:KeyName>` elements in the `<KeyDescriptor>` elements in the Metadata. The rules for this can get fairly complex, and the opportunities for screwing this up are numerous, which is one reason we don't recommend this approach.

The security of the system also demands that you understand the issuance policies of the CAs that you rely on, and how their certificate naming policies match up with the lines you want to draw around your SAML deployments. In other words, if IdP "Foo" is allowed to use the certificate named "Bar", what prevents somebody else from getting a certificate named "Bar" that's also issued by one of the CAs you accept? Deployments relying on multiple CAs can be tricky to keep safe. Using a single CA is a bit more comprehensible.

Finally, you have to take account of the fact that some of the information in the metadata besides just the key information has security implications, including the fact that particular endpoints are controlled by the same system that you're assigning certificate names to. This creates additional complications when you think about the security of the metadata and how you deal with key compromise, because it may not just be the key that you care about. If you end up deciding that you have to get the metadata as a whole revoked, then essentially nothing's been gained by managing the key's validity separately. All you've done is create two parallel infrastructures that have to be linked in a complex way.

Dynamic PKIX

The usual approach is to express the set of trust anchors to validate end-entity certificates against dynamically, using a `<shibmd:KeyAuthority>` extension in the Metadata. The extension is placed at or above the level of a given entity, such that you can express a set of trust anchors to apply to one or more entities, or different anchors for different entities, rather than a one-set-fits-all model as one sees in most software.

The benefit is that the metadata is still self-contained in terms of expressing the information, but you also have to make sure that you have CRLs (if any) embedded in the `<shibmd:KeyAuthority>` extensions you use, because that's the only supported mechanism for preventing use of a revoked certificate when using this plug-in.

Note that since remote CRLs aren't supported (in the context of this plug-in), and the CRLs for the large commercial CAs are really too large and unmanageable to include, this effectively precludes the use of commercial PKI within your deployment when this plug-in is used. The "static" version is a better fit.

Static PKIX

A approach that requires additional local configuration is to specify the set of trust anchors to validate end-entity certificates against directly as a set of certificates and CRLs. This usually involves a set of local files containing the CAs to accept and a set of remote URLs pointing to the CRLs to use.

Metadata Distribution and Verification

Because the models described above all rely on metadata as a critical (or the only) input to making trust decisions, the security of the system usually has a lot to do with how the metadata is actually delivered and verified by the IdP and SP. The specific threats and mitigations vary between the different models, which creates a lot of opportunities for mistakes.

The sets of protections that are available to mix and match can largely be enumerated as follows:

- XML Signatures over the Metadata
  - Metadata is often digitally signed so that the integrity of the metadata can be maintained across time and without relying on a secure delivery path. A range of traditional PKI controls can be applied to the validation of the signing key.
- Expiration
  - A `validUntil` attribute can be placed into the metadata to limit the time for which the information can be accepted and considered valid.
- Use of SSL/TLS
  - Metadata can be hosted on a secure web site and only used when acquired from a trusted server endpoint. A range of traditional PKI controls can be applied to the validation of the server certificate.
- Cache Control
  - A `cacheDuration` attribute can be placed into the metadata to limit the time for which the information should be kept without contacting a trusted source to update it.

Broadly speaking, the first two and last two tend to go "together" as part of a strategy to control the acceptance of metadata and limit the effect of a compromise of the metadata, or the key material inside it. The actual limitations here have more to do with what a particular piece of software can handle; people can get extremely creative, but the more you try to get cute, the less likely you'll end up with something reliable or secure.

Securing the "Inline" Model

Ignoring certificates at runtime and using metadata as a kind of "pseudo-certificate" has a lot of benefits, but in return for these advantages, deployers are required to move the "risk mitigation" mechanisms found in an X.509 PKI, such as revocation lists, to the metadata layer instead.

Of course, a brute-force approach is to treat revocation as a manual process. If you're comfortable with every system's administrator having to be contacted if the metadata changes or a key is compromised, then you can certainly take the easy way out and sign a metadata file with no `validUntil` attribute or use a window of months or years. Just understand that this is essentially treating any revocation event as a major disruption, with no means of preventing a compromised key from being used essentially indefinitely. This is not advisable, and the PKIX model is probably a better fit for your needs, but in turn requires that you understand X.509 PKI fairly well.

"Sign and Expire"

Metadata files usually need to be signed and carry a `validUntil` attribute that will expire the information according to a reasonable timetable that's no longer than the usual interval one might use for a CRL, typically on the order of a few weeks at most. As a consequence, the metadata also needs to be re-signed periodically.

To see why, consider that a metadata file containing a compromised key but no `validUntil` attribute would be accepted by any software that trusted the metadata signing key, effectively making it impossible to revoke the compromised key except by revoking the metadata signing key. This is not helpful. The Shibboleth software includes metadata "filters" that can enforce the presence of the `validUntil` attribute, and these filters should be used in conjunction with this approach to ensure that signed metadata that omits any upper bound on use can't be accidentally introduced into a deployment.

Deployments are also expected to frequently refresh the metadata they rely on, usually daily. The purpose of the `validUntil` attribute is not to substitute for this daily refresh but to limit the damage that revoked metadata information can cause. By updating often, the actual window of exposure can be very limited, generally to something between what a PKI relying on CRLs and one relying on OCSP could achieve.

"Download and Cache"

An alternative approach to the "sign and expire" model is to obtain metadata in real-time from a trusted endpoint, and cache it for short periods (usually based on the inclusion of a `cacheDuration` attribute in the metadata). The thing to remember about this approach is that it relies on the fact that metadata has to be acquired directly from a trusted endpoint (using SSL/TLS) if it doesn't carry a `validUntil` attribute. This approach is effectively much like an OCSP-based PKI.

Metadata Key Verification

The actual verification of the signing key or SSL/TLS endpoint can be accomplished in a lot of ways. Essentially this is where the traditional PKI notions get reintroduced, but they secure the metadata instead of securing individual transactions. Some deployments may find it sufficient to distribute a signing key out of band, much like a CA, and if there's a total compromise, treat it as a rare, drastic event. This is particularly reasonable if the metadata signing key, like many CAs, is only accessible offline and protected by physical controls.

On the other hand, if your signing key is online (which has substantial operational benefits), you may well choose to ensure that you can safely revoke that key if it's compromised by indirecting the validation of that key through a PKI in the traditional way; you issue a certificate for the metadata key using an offline CA, and deploy that CA key (along with a CRL location) to all of the sites as the root of the verification process.

Securing the "PKIX" Model

It's difficult to assess exactly what the various risks are with this model, which is a big reason why it's not recommended anymore. It's certainly the case that you can take an aggressive stance and handle your metadata business just as with the "Inline" model above, because that effectively creates a parallel PKI-like set of constraints on metadata validity that can't make your systems any less secure than if you didn't. But it's also not clear in that case that you've gained anything.

The main advantage to this approach is that you get something a bit more palatable to commercial SAML software, but that assumes you understand what that software is doing well enough to understand where the gaps will be with its handling of metadata, and you'll certainly have to configure the trust anchors in a manner that's separate from the metadata. This might fail outright if you're trying to use different trust anchors for different IdPs or SPs you define to the product.

If you were to try to relax the controls that are discussed above and somehow "do less" on the basis that the PKI in place to validate the certificates for you, then you have to think through the implications of various kinds of changes or compromises to see what the problems might be. In general, there are subtle exposures that can arise in relation to the bindings between entities and certificate names, and X.509 PKIs aren't really designed with this indirection in mind. This impedance mismatch is the biggest reason we advise against it, as it makes reasoning about the system very difficult.

The point here is that even if you think the actual keys are being managed effectively and can be revoked when needed, you still have to consider the implications of the rest of the metadata and whether you might need to revoke the entity/key name binding, and if so, X.509 won't help you with that.