Message Level Security

A SWITCH representative has made a case for enabling a number of SP options by default to enable more widespread use of message-level security in place of transport (TLS) security for SOAP-based profiles. The general justification for this is a claim that supporting an alternate port for SOAP traffic, particularly with client TLS enabled, is more complex than the alternatives.

Historically this view is consistent with a lot of older vendor practices in the Liberty community, which favored message signing over client TLS authentication. Unfortunately there are a number of software compatibility issues due to the original Shibboleth preference for TLS that make a wholesale change a difficult prospect, along with a number of non-obvious issues connected with such a shift.

This document is not so much an advocacy statement for one approach or the other, but a relatively unbiased set of considerations to be used as input to future decision-making. It's quite clear that changing defaults in this area is a major-version undertaking.

As a general matter, in each use case, the focus is on two aspects of security, authentication and confidentiality. Message integrity is normally part of providing message authentication, and authentication is a precondition for confidentiality.

Front-Channel

The original request to look at this issue was not made in regard to the front-channel use of signatures, but to SOAP. However, it's worth a brief mention because it's a simpler case.

Today, we default to no security measures in requests to the IdP and a standard set of defaults for IdP responses that vary by SAML version. A significant advantage to signing SP requests is the elimination of endpoint registration and checking for SAML 2.0 SSO. This is particularly valuable for vhost or gateway scenarios in which many endpoints would need to be registered.

The downside is a potential for denial of service attacks by an unauthenticated remote attacker by triggering a high number of message signature operations. Most SSL protected sites are already vulnerable to this kind of risk from an attacker than forces handshakes, but this lowers the burden for such an attack. A possible mitigation could be to implement throttling by IP address, which would also help address inadvertent looping problems.

For this to make sense, we would also need to see IdPs ship with the endpoint bypass logic enabled by default, which would not happen until 3.0 at the earliest. In practice, if an SP wanted to avoid registering endpoints in metadata, the cost of turning on a simple option is insignificant, and so the SP default here is fairly unimportant. It's also possible to trigger signing by the SP by means of the wantAuthnRequestsSigned flag in metadata.

Summing up:

- Goal: Avoid endpoint registration.
- Deployment Constraint: All IdPs have to support bypassing the endpoint check (and of course must support signed requests to begin with).
- Likely Scenario: Useful only in scenarios where IdPs are known to support the feature, insignificant cost to enable signing in the SP, some risk in changing the default, so unlikely to make sense.
- Question: Should we implement any DoS mitigation in the SP?

Back-Channel

The SOAP-based profiles are where the bulk of the complexity lies. The frustrating conclusion is that there's essentially no functional advantage to avoiding mutual TLS on a dedicated port; it's by far the fastest, simplest to understand, easiest to implement, and most secure approach to the problem. So, not a huge surprise that we picked it.

What it does have, though, is some deployment costs:

- Many deployers are clueless in basic web site administration or have problems supporting extra ports because of firewall issues.
- Only a handful of deployers have any experience configuring TLS client authentication.
- Using a single port and virtual host usually means configuring renegotiation-based TLS in a subdirectory, which has been a large source of web server bugs.
- Relying on server authentication (and confidentiality) via a TLS certificate issued by a browser-friendly CA creates significant burdens for managing trust. SPs must have mechanisms for handling frequent certificate rollover by the IdP, and we all know that essentially everything but Shibboleth and simpleSAML.php don't.
- Supporting TLS client authentication usually negates the off-loading of TLS to load balancers, and even then, there are substantial bugs and limitations involving certificate validity and chain size caused by bugs in Apache and some Java containers. Avoiding Apache here altogether tends to help.
- The vast majority of commercial implementations fall down when it comes to doing non-traditional trust management of TLS connectivity, even if they support alternatives for message signing. Basically, they can't be bothered to do anything but call into the HTTP stack's TLS support and let it do whatever it wants to do.

Breaking Down the Model

To get into a discussion of what could be changed, we have to break down the pieces of the security problem that we're solving with mutual TLS, because we either have to reimplement each piece or agree that a loss of function is acceptable.

- Request Authentication
- Request Confidentiality Guarantee to Client
• Server-Side Protection from MITM
• Response Authentication
• Response Confidentiality Guarantee to Server
• Client-Side Protection from MITM

In this breakdown, I include message integrity guarantees as part of message authentication, because the technology for one generally supplies the other "for free". However, note that confidentiality protection is directional (the sender of a message needs a guarantee that its message can be confidential).

MITM protection is related to the other properties, but the possibility of a MITM raised by avoiding mutual TLS may incur costs in their pursuit.

Request Authentication

The alternative to client TLS is to digitally sign the SAML request. The SOAP layer is generally ignored here, though technically the message could be signed at that level, and would need to be in some specialized cases.

It is generally believed that most commercial implementations support this for verifying requests. Shibboleth does since version 2.0, but older versions only included limited support for this in the case of artifact requests. Signed attribute queries were not supported in any 1.3 releases.

From a compatibility point of view, it's likely that we could enable signed requests by default without incurring major risk, by leaving client TLS active at the same time. Non-supporting implementations should ignore the signature, and there should be no overhead with IdPs that turn off client TLS, since the handshake just won't include the client certificate.

The only downside would be some overhead in unnecessary signing, but SOAP traffic is relatively rare and not triggered by unauthenticated activity. Mitigating this would require work on metadata extensions for attaching security policy to endpoints.

Summing up:

• Goal: Allow for IdPs to ignore/disable client TLS.
• Deployment Constraint: Legacy IdPs and probably other commercial variants won't support signatures, so client TLS would need to remain active.
• Likely Scenario: Seems like enabling it by default could be risk-free, and exist alongside client TLS, but this would need to be tested.
• Question: Should we shoot for changing this in V2.5, or wait for a 3.0? Should we develop metadata extensions to guard against interop issues?

Request Confidentiality Guarantee to Client

Confidentiality is typically just assumed because of server TLS authentication, and the SP currently implements a flag on SOAP requests that enforces confidentiality by checking for TLS. This is actually a bug, since it means that TLS is assumed to provide authentication of the server as well, and the setting would be broken if TLS were in place but the server wasn't authenticated.

The flag for this check also defaults on, so if the bug were fixed to limit the definition of confidentiality at the transport layer to cases where the transport provided authentication, exceptions to this would break unless the flag were manually turned off.

To achieve confidentiality outside of the transport layer, some encryption of sensitive content is possible (EncryptedID), but there are exceptions to this (e.g., artifacts can't be encrypted). The IdP also doesn't currently support EncryptedID.

The problem is that to continue to rely on the transport for request confidentiality, we must of necessity rely on the transport for response authentication. This leads directly to the problem of deployers backing into the use of commercial certificates for SOAP endpoints, if they want to gain the advantage of eliminating the extra port/vhost.

For Shibboleth, this is a tractable problem because we have metadata-based key management, but deployers are very likely to forget the fact that their meaningless TLS certificate has to be rolled over in metadata ahead of renewing it on their web server. And there's still the problem of commercial implementations, which will never support key management of any kind other than pseudo-PKIX hackery. But it must be admitted that in general, use of SOAP is likely to be limited from non-Shibboleth SPs. One exception: EZProxy.

Conclusions: nothing very positive here. I think the use of port 443 for SOAP is a fairly bad idea that will cause immense pain in most cases. Clearly there will be exceptions, but in such cases, I'd expect server TLS to continue to be used for authentication anyway.

Server-Side Protection from MITM

On the server end, the use of client TLS allows the server to ignore the risks of a MITM attacker. Using message signatures creates the challenge that while you know the origin of the message, you don't actually know the sender. An attacker could be in the middle, which raises the spectre of needing to ensure confidentiality for responses somehow, and the need to be careful with what you hand back to the sender, since it could be an attacker.

The usual mitigation for this is freshness and replay detection. This does limit the possibility of a passive attacker replaying old messages, but within the replay window, you have the risks that the replay cache might not be reliable enough or that the message is simply intercepted.

It turns out there's a better way to handle this: channel binding. The goal is to authenticate at the message layer, but bind that to the transport layer (TLS) so that you can leverage that layer as well. This turns out to be fairly simple when you're trying to solve this specific problem (and the related problem of confidentiality guarantees for the server).

Normally channel binding comes in "unique" and "endpoint" flavors. The former is virtually impossible to implement, but allows the parties to bind to the exact TLS session in use. The latter is limited to verifying that both parties know the server's TLS certificate, and is more complex to use when multiple connections are involved in a session, but is simple to use across a single exchange, as in the SOAP over HTTP case.

Basically, we have the client include the server's certificate (or its hash) in the signed request (assuming its verified with a TLS handshake), and the server verifies that its certificate or hash is in the request, verifying the channel. At that point, the other end of the TCP connection must have been connected to the server directly, or it wouldn't have seen that certificate at the other end of the TLS connection. Thus, the server knows that a correctly implemented client is in fact the sender of the message, and there is no MITM.

Summing up:
- Goal: Protect from MITM attacks without client TLS.
- Deployment Constraint: Replay/freshness checks should be in place, but channel binding would require new code (not to mention finishing that spec).
- Likely Scenario: We live with things as is given the low risk, but add channel binding.

**Response Authentication**

As in the request case, it is possible to substitute digital signing of the response for server TLS. In this case, TLS may well be in place, but the certificate may not be trustworthy, and the signature can rely on a different key.

The degree of support for this in commercial products is much less clear than request signing. In theory it isn't any different, but in practice I would be surprised if it were supported very widely. I don't believe it was supported by the 1.3 SP, but this would need to be verified. It is supported in 2.0.

The use case for this is very clear; as noted above, commercial TLS, let alone PKIX, makes a horrid basis for SAML trust, and avoiding the verification of those certificates is a huge win. This makes a single port IdP practical.

As a compatibility matter, one would assume that an unsupported signature will usually be ignored, but this would create a requirement to simultaneously allow for verification of either the signing key or the TLS certificate depending on what the client can handle.

Summing up:

- Goal: Allow for IdPs to run on a single port.
- Deployment Constraint: Legacy SPs and probably other commercial variants won't support signatures, so server TLS would need to remain active, and more importantly, trustable.
- Likely Scenario: Seems like enabling it by default could be risk-free, and exist alongside server TLS, but this would need to be tested. In constrained scenarios, it may be possible to leave the TLS certificate out of the metadata, but not in general.
- Question: Should we make single port deployment the primary 3.0 model?

**Response Confidentiality Guarantee to Server**

Again like the request case, the server normally relies on the presumption of mutual TLS to establish confidentiality. My guess is the IdP is like the SP and treats any TLS connection as sufficient to avoid the need for encryption (i.e., the conditional option). This breaks if the request is signed instead.

It is however the case that the important data can be encrypted and the IdP supports that already for the most part, so simply altering the controls over that decision may be enough to address this problem.

Channel binding, discussed earlier, can also factor into this equation. If a response is being returned over a connection on a bound channel, TLS can be leveraged even if the client end isn't authenticated.

But in the meantime, we may have bugs/holes in the determination of whether to encrypt if we were to see IdPs deployed without client TLS enabled, which may mean waiting to recommend such an approach until they're fixed (probably until 3.0).

Summing up:

- Goal: Guarantee confidentiality to client without client TLS.
- Deployment Constraint: The IdP needs to know that confidentiality isn't available if client TLS isn't used.
- Likely Scenario: We add channel binding and advise deployers wanting to support this in the meantime to force encryption.
- Question: Does the IdP (like the SP) assume that the presence of TLS alone guarantees confidentiality?

**Client-Side Protection from MITM**

Finally, we again see that protecting against MITM without verifying the server's certificate requires some imperfect applications of freshness and replay detection to limit the possibility that a response is actually coming through an attacker.

We can also leverage channel binding for this, because it's after the fact. If the server includes its channel binding in the signed response, the client can compare and know that it's communicating directly with that server.

However, this doesn't work for confidentiality on the request, since the determination isn't made until we see the response. Fixing that would require an additional round trip and isn't compatible with implementations.

Summing up:

- Goal: Protect from MITM attacks without server TLS.
- Deployment Constraint: Replay/freshness checks should be in place, but channel binding would require new code (not to mention finishing that spec).
- Likely Scenario: We live with things as is given the low risk, but add channel binding.