Engineering Attestable Services (short paper)

John Lyle and Andrew Martin

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD
{john.lyle,andrew.martin}@comlab.ox.ac.uk

Abstract. Web services require complex middleware in order to communicate using XML standards. However, this software increases vulnerability to runtime attack and makes remote attestation difficult. We propose to solve this problem by dividing services onto two platforms, an untrusted front-end, implementing the middleware, and a trustworthy back-end with a minimal trusted computing base.

1 Introduction

Web services are a popular way of implementing component-based systems. They have a number of potential advantages, offering higher reliability and integrity due to component reuse and dynamic selection. However, some have significant security concerns, such as those in healthcare and financial scenarios. To fulfil these security requirements, mechanisms are needed to gain assurance in the platforms hosting these services.

One method for assessing a platform is attestation, part of the functionality provided by Trusted Computing. This allows a remote party to find out the exact software configuration being used. If all the software running at a service is well known and trustworthy, then the user can potentially trust it. However, web services use a great deal of complicated software, and little of it may be considered trustworthy. Runtime attacks also remain possible, making remote attestation an impractical solution [1].

The complexity of service middleware, and its position in the trusted computing base (TCB) of a web service, is a significant part of the problem. Service providers require the middleware to provide features such as load balancing and monitoring, along with parsers for complex languages like SOAP. All the libraries that implement these features are of little interest to the end user, but are still part of the TCB and must be attested. This makes it impossible to guarantee the integrity of the service, or the confidentiality of data sent to it, as it all relies on untrustworthy middleware.

The solution we propose is to divide the web service middleware and logic onto different platforms. The middleware platform is then free to implement functionality that the service provider cares about, but remains untrusted by the end user. The integrity of the service application is guaranteed by the second platform, which has a much smaller trusted computing base and is less vulnerable to runtime attack. This makes the service more trustworthy and attestation more practical.

1.1 Trusted Computing

Trusted computing is a paradigm developed by the Trusted Computing Group [2]. It aims to enforce trustworthy behaviour of computing platforms by securely identifying all hardware and software that it uses. If a platform owner can find out what software and hardware is in use, they should be able to recognise and eliminate malware.

The technologies proposed by the TCG are centred around the Trusted Platform Module (TPM). In a basic server implementation, the TPM is a chip connected to the CPU. It provides isolated storage of RSA keys and Platform Configuration Registers (PCRs). These PCRs can be used to hold *integrity measurements*, in the form of 20 byte SHA-1 hash values. They can only be written to in one way: through the `extend` command. This appends the current register value to the supplied input, hashes it, and stores the result in the PCR. In order to work out what individual inputs have been added to a PCR, a separate log is kept. When this log is replayed, by rehashing every entry in order, the final result should match the value in the PCR.

The limited functionality offered by the TPM can be used to record the boot process. Starting from the BIOS, every piece of code is hashed and extended ('measured') into a PCR by the preceding piece of code. This principle is known as *measure before load* and must be followed by all applications. If so, no program can be executed before being measured, and because the PCRs cannot be erased, this means that no program can conceal its execution from the TPM. A platform is said to support *authenticated boot* when it follows this process.

1.2 Remote Attestation

The TPM allows a platform to report integrity measurements through *remote attestation.* When challenged, the TPM can create a signed copy of its PCRs. This is used by a remote party to verify the platform's measurement log. PCRs are signed using a key held by the TPM, guaranteeing its confidentiality. This Attestation Identity Key (AIK) is certified by an authority (a 'Privacy CA') [2].

The software running at the platform can be identified by matching the hash values in the measurement log with reference data. This requires a list of *reference integrity measurements* (RIMs) contained within a Reference Manifest Database [2].

1.3 Protecting Data and Keys

The TPM can be used to encrypt data and only allow decryption when PCRs are in a predefined state. TPM RSA keys can be created so that they are *bound* to PCR values through the `CreateWrapKey` command. The private half is then always held securely in the TPM. When it needs to be used, a request ('`unbind`') is made to apply the private key to the encrypted data. The TPM will only complete the request when the PCRs are in the state defined upon key creation. A credential for the bound key, certifying that the private-half of it is held in the

TPM and restricted to certain PCRs, can be generated (using an AIK) through the TPM's `CertifyKey` operation.

1.4 Why are Web Services Difficult to Measure and Attest?

Attesting a web service is difficult in practice. The amount of software to measure is surprisingly large – in recent work [3], we found that a typical web service made around 300 integrity measurements, and that, on average, 35 new RIMs were required for updates every month. This is a potentially impractical quantity of software to test and evaluate.

The large TCB is partly due to functional and interoperability requirements. High-level communication protocols [4] used by services require complicated software to process. Servers also have many sophisticated features dedicated to internal requirements such as auditing and management. These are important to the service provider, but not the requester, and yet all must be reported in an attestation. Most operating systems are also guilty of having a large code base, and provide relatively weak isolation. This makes the system error-prone and vulnerable to compromise. Attestation is therefore less valuable, as the chance that a successful runtime attack has been performed is high. Minimizing the trusted computing base appears to be essential.

One component to minimize is middleware. In our experiments, removing it resulted in a 30% fewer integrity measurements [3]. The popular Glassfish application server has around 300 modules (some optional) totalling nearly 100 megabytes of compressed bytecode. Furthermore, middleware is responsible for parsing complex data structures and processing input, obvious targets for attack. Removing it would also reduce the number of features that the operating system has to support, potentially improving efforts to minimize the OS runtime footprint. We believe that this makes a compelling case for removing middleware from the trusted platform. However, middleware provides essential functionality, and it cannot be removed altogether. The next section discusses how to move it away from the TCB without losing any functionality.

2 Removing Web Service Middleware from the Trusted Computing Base

We propose that web services can be deployed so that they support heavyweight protocols and features but have a small TCB. This is achieved by divided them into two components, one trusted and one not. The untrusted component acts as a proxy, and is the perceived endpoint for all web service interactions. It communicates with the outside world through SOAP and XML and performs management functions such as load balancing and auditing. The trusted back-end server provides all the real functionality and logic. In a data processing scenario, the back-end platform could either be a data store, or be responsible for contacting it and forming queries. Communication between the front and

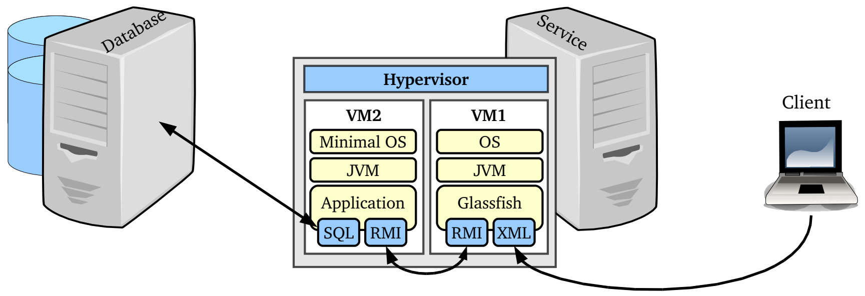

Fig. 1. The split service architecture. Lines show message flow.

back-end is through a simple protocol that requires a less-complex parser, such as Java RMI. Figure 1 illustrates this system.

The advantage of this architecture is that the back-end can attest to a simple configuration. It can also use a minimal operating system, perhaps even a bytecode processor. Furthermore, it only needs to parse input from one protocol, and XML does not need to be interpreted. Attestation should therefore be appropriate. Of course, the back-end server has been intentionally designed to not require a web service stack, and therefore attestations must be proxied by the front-end. The rest of this section discusses additional steps and modifications required to realise this proposal.

2.1 Establishing a Secure Channel

Assuming the back-end service is trusted, the next step is to guarantee a secure channel. This is a challenge, as the front-end is proxying all traffic. A *platform in the middle attack* [5] must be avoided, so that the platform that originally attested is the same one that we are then sending requests to. Solutions using transport-level encryption have been discussed before [6], but in our scenario we cannot use TLS with a key held on the back-end, as this would prevent the front-end platform from translating and forwarding requests. Instead, we use message-level cryptography [7]. To do this, the back-end can publish a public key, along with a certificate generated by the TPM's `CertifyKey` command. If the same AIK were used for the attestation process, this establishes that the key belongs to the attested platform. Furthermore, if the key is bound to known-good PCR values, this key can guarantee platform state.

An initial request for a service's public key can follow the WS-Trust specification. The protocol below shows the user (U), credential repository (C), service (S), service public key (S_{pub}) and service AIK (S_{aik}). Line 1 is a request for a service's public, bound TPM key, and line 2 is the response, containing a service key and TPM credential, signed by service's AIK. These steps must be performed in a transport session with a known, trustworthy credential repository:

$$U \rightarrow C : \quad \text{RequestSecurityToken},\ S \tag{1}$$

$$C \rightarrow U : \quad S_{pub}\ ,\ S_{aik}\ ,\ \{S_{pub}, \text{TPM\_CertifyInfo}\}_{S_{aik}} \tag{2}$$

Service requesters can use this public key to encrypt messages without fear of loss of confidentiality. Furthermore, any reply message generated by the endpoint can be signed, proving the source of the reply. We propose the following protocol, with the service front- and back- ends denoted as F and S respectively, using an encrypted session key:

$$U \rightarrow F: \quad \text{Method}(\ \{nonce_U, arg_1, arg_2...\}_K\ ), \{K\}_{S_{pub}} \quad \text{(SOAP)} \tag{3}$$

$$F \rightarrow S: \quad \text{Method}(\ \{nonce_U, arg_1, arg_2...\}_K\ ), \{K\}_{S_{pub}} \quad \text{(RMI)} \tag{4}$$

$$S \rightarrow F: \quad \text{Reply}, HMAC(nonce_U,\ \text{reply}\ )_{S_{pub}} \quad \text{(RMI)} \tag{5}$$

$$F \rightarrow U: \quad \text{Reply}, HMAC(nonce_U,\ \text{reply}\ )_{S_{pub}} \quad \text{(SOAP)} \tag{6}$$

Line 3 is the SOAP method invocation with session key K applied to all field, which is then translated and forwarded via RMI in line 4. The reply is generated in line 5 and translated again to conform to WS standards in line 6. If the TPM key S_{priv} is not bound to PCR values, then an additional WS-Attestation step is required first, which also must be proxied by the front-end.

2.2 Preserving Integrity and Confidentiality

The messages described in lines 3 to 6 of Section 2.1 is simplified in terms of signatures and encryption. Decryption of incoming messages, and signing of the result, must be performed on the back-end, as only it has access to the S_{priv} key. However, this means that only individual fields can be encrypted, not complex XML structures, as the back-end cannot process XML. An attacker now has the opportunity to re-order fields, as nothing binds the content of the field to its location in the document. If the encryption is just of the field itself, then it will also be vulnerable to replay, as no freshness information is present. The same is true for the signed response message from the back-end platform.

To provide both freshness and structure to the elements, without breaking web service standards, fields must be added to the internal methods and the response. The response should contain a hash of the original input, result and a nonce. To avoid the endpoint from needing to process XML, we suggest that a set of identifiers be included internally, linking the expected XML structure to the internal fields. The identifier-result structure is then signed by the endpoint, and included in the response. The example in part 4 of Figure 2 demonstrates this. The verifying party can then compare the request and result against the arguments and result the endpoint believes it has used and computed. We have used XPATHS as IDs, noting that these should be predictable and easy for the verifier to process.

3 Security Analysis

Demchenko et al. [8] and Bhalla and Kazerooni [9] identify threats to XML services. These include misuse of user credentials, unencrypted SOAP messages, maliciously formed input, XML parsers exploits, WSDL enumeration, poor site configuration and error handling. Our proposals reduce the impact of some of these issues, in comparison to a standard Web Service endpoint that also uses message-level encryption.

```
1) Original SOAP Request
<soap:body ... >
 <m:Entry>
  <m:from>Joe Bloggs</m:from>
  <m:content>...</m:content>
  <m:nonce>36829463846238</m:nonce>
 </m:Entry>
</soap:body>
```

```
2) Encrypted SOAP Request
<soap:Header>
 <wsse:Security><xenc:EncryptedKey>
    <ds:KeyInfo ... >
     <ds:KeyName>PubKey X</ds:KeyName>
    </ds:KeyInfo>
    <CipherData><CipherValue>
       [Encrypted Symm Key]
    </CipherValue></CipherData>
    <ReferenceList>
      <DataReference URI='#content'/>
      <DataReference URI='#name'/>
    </ReferenceList>
    <CarriedKeyName>EndpointKey
    </CarriedKeyName>
  </xenc:EncryptedKey></wsse:Security>
</soap:Header>
<soap:Body><m:Entry>
 <m:from>
  <xenc:EncryptedData Id="name">
  <xenc:CipherData><xenc:CipherValue>
       [Encrypted Name]
  </xenc:CipherValue></xenc:CipherData>
  </xenc:EncryptedData>
  </m:from>
  <m:content>
    <xenc:EncryptedData Id="content">
       ...[Encrypted Content]...
    </xenc:EncryptedData>
  </m:content>
  <m:nonce>36829463846238</m:nonce>
</m:Entry></soap:Body>
```

```
3) RMI Request
response = endpoint.submit(
  [encSymmKey],     // enc. session key
  "Pub Key X",      // TPM key ID
  [Enc Name],[Enc. Content], //fields
  36829463846238 ); // nonce
```

```
4) ASN.1 style response structure
messageInfo MessageInfo ::= {
  input {
    encrypted-symm-key  [encSymmKey],
    pub-key-id            Pub Key X  ,
    variables  {
      { field-xpath     //m:Entry/m:from  ,
        field-value   [Encrypted Name] },
      { field-xpath     //m:Entry/m:content
        field-value   [Encrypted Content] },
      { field-xpath     //m:Entry/m:nonce  ,
        field-value   36829463846238 }},
  result {
      { field-xpath  //m:Response/m:Success,
        field-value  1 }}}
```

```
5) RMI Response
return new MessageResponse (
  result,
  messageInfo,
  SHA1( messageInfo ),
  Sign( SHA1( messageInfo ) )
  // signed with endpoint private key );
```

```
6) SOAP Response
<soap:Header> ...
  <Signature ... >
   <ds:Signature ... >
    <ds:SignedInfo>
     <ds:Reference URI="#MsgVerification">
      <ds:DigestValue>[SHA(messageInfo)]
      </ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>
     [Sign( SHA1( messageInfo ) )]
    </ds:SignatureValue>
   </ds:Signature>
  </Signature>
</soap:Header>
<soap:Body ... >
  <m:Response>
   <m:Success>1</m:Success>
    <m:Verification id="MsgVerification">
     [messageInfo]
    </m:Verification>
   </m:Response>
</soap:Body>
```

Fig. 2. Service request and response transformations

Threats from SOAP parsers are eliminated in this architecture, as they can only compromise the front-end. These threats are significant as several attacks have been published on XML parsers[1]. Of course, vulnerabilities in the parser used to communicate between front and back-end components would still have an impact, but the protocol is less complex, and few vulnerabilities in Java RMI (for example) have been published. Similarly, vulnerabilities in application servers, such as Glassfish and Apache Axis 2, would have a much smaller impact in our system.

Use of poorly-configured services can be avoided through use of remote attestation. This is true of any attestation-enabled platform, but our architecture reduces the number of components to report upon, thus reducing complexity and making it easier for a verifier to establish the properties required. Long-term credentials can also be stored safely using a TPM, reducing this vulnerability.

[1] For example, Secunia Advisories SA22333 and SA10398

However, though the front-end service may be untrusted, it can still impact availability, resulting in a denial of service attack. As we have only *split* the service into two components, rather than increasing the amount of software, this is no worse than before our modifications. The same is true of error handling.

4 Performance

The proposed architecture will have a performance overhead due to additional RMI requests and TPM operations. Gray [10] provides a performance comparison of RMI and Web Services. His figures show that RMI invocations take around 1ms and are therefore an order of magnitude faster than most WS-Security enabled web services. We would therefore expect the additional RMI step to have a negligible impact on round-trip time. Furthermore, should the front- and back-end services be hosted on the same platform (such as in Figure 1) then we can be even more optimistic.

The impact of using the TPM is more significant. For each message, the TPM must decrypt a symmetric key using a key bound to the TPM, and then sign a digest using another bound key. With an Infineon 1.2 TPM, these operations take 400ms each, addding 800ms to the round trip time. A faster alternative would be to use the same session key repeatedly for the service, which would eliminate subsequent unseal operation on messages received from the same client. The session key could also be re-used for signing, meaning only one TPM operation in total. The disadvantage to doing this is that the key is stored in unprotected memory for a significant period of time. Further optimisation may be possible with virtual TPMs, operating mostly in software.

5 Related Work

Wei et al. [11] split web service middleware into trusted and untrusted parts. Sensitive information in incoming messages is intercepted by a 'message splicer' and only given to the trusted module. This is similar to our solution, but we take the proposals further, allowing users to attest, rather than just hardening the internal structure. Our proposals solve the problem of trusting the server-side message splicer.

Similarly, Jiang et al. [12] mitigate the threat from malicious insiders by using an IBM 4758 secure co-processor. This 'guardian' is responsible for some important functions, and users can establish a secure session directly with it. Our approach expands on this is two ways: allowing conformance with service standards and using a low-cost Trusted Platform Module. Furthermore, our system is designed to minimise threats from both outside and insider attackers.

Watanabe et al. [4] have an alternative approach, separating the communications component - the 'Secure Message Router' - from the application itself. This SMR is a trusted component. This is the opposite of our proposal, and focuses on establishing guaranteed secure communications, rather than service integrity.

This might be a way to implement *composite* services, which our architecture does not allow.

6 Conclusion

We have shown that web service middleware is a significant limiting factor in attestation and establishing trustworthiness. Our proposal is to remove it from the trusted computing base of the service, solving both problems and increasing resilience to runtime attack. The back-end platform can then be used to run verified services with critical functionality. We have also outlined a method for establishing a secure session without sacrificing web service standards. From analysis of security benefits against performance overhead, we believe that this architecture is worth considering for any web service with known, high security requirements.

Acknowledgements

John Lyle's work is funded by a studentship from QinetiQ and the EPSRC. We thank Andrew Cooper, Jun Ho Huh and Cornelius Namiluko for their suggestions.

References

1. Schellekens, D., Wyseur, B., Preneel, B.: Remote Attestation on Legacy Operating Systems With Trusted Platform Modules. ENTCS **197**(1) (2008) 59–72
2. The Trusted Computing Group: Website (2009)
3. Lyle, J., Martin, A.: On the feasibility of remote attestation for web services. In: SecureCom '09. Volume 3. (2009) 283–288
4. Watanabe, Y., Yoshihama, S., Mishina, T., Kudo, M., Maruyama, H.: Bridging the Gap Between Inter-communication Boundary and Internal Trusted Components. In: ESORICS. Volume 4189 of LNCS., Springer (2006) 65–80
5. Bangerter, E., Djackov, M., Sadeghi, A.R.: A demonstrative ad hoc attestation system. In: ISC. Volume 5222 of LNCS., Springer (2008) 17–30
6. Gasmi, Y., Sadeghi, A.R., Stewin, P., Unger, M., Asokan, N.: Beyond secure channels. In: STC, New York, NY, USA, ACM (2007) 30–40
7. OASIS: Web services security: Soap message security 1.1. `http://docs.oasis-open.org/wss/v1.1/` (2004)
8. Demchenko, Y., Gommans, L., de Laat, C., Oudenaarde, B.: Web services and grid security vulnerabilities and threats analysis and model. In: GRID, IEEE (2005)
9. Bhalla, N., Kazerooni, S.: Web service vulnerabilities. `http://www.blackhat.com/presentations/bh-europe-07/Bhalla-Kazerooni/Whitepaper/bh-eu-07-bhalla-WP.pdf` (2007)
10. Gray, N.A.B.: Comparison of web services, java-rmi, and corba service implementation. In: Australasian Workshop on Software and System Architectures. (2004)
11. Wei, J., Singaravelu, L., Pu, C.: A secure information flow architecture for web service platforms. IEEE Trans. on Services Computing **1**(2) (2008) 75–87
12. Jiang, S., Smith, S., Minami, K.: Securing web servers against insider attack. In: ACSAC, IEEE (2001) 265