How to Trust a Web Service Monitor Deployed in an Untrusted Environment?

Mahantesh Hosamani Harish Narayanappa Hridesh Rajan
Dept. of Computer Science, Iowa State University
{mahantesh, harish, hridesh}@cs.iastate.edu

Abstract

In a service oriented architecture, certain requirements can be tested by observing the interface of the service whereas other requirements such as data privacy, confidentiality and integrity cannot be tested in this way. After deployment, a requirements monitor is used to analyze the conformance of a web service to such requirements. The integrity of the reported conformance results is as good as of the integrity of the monitor especially when the requirements monitor is executing in an untrustworthy environment. In this paper, we propose a hardware-based dynamic attestation mechanism to validate the integrity of the requirements monitor. To evaluate our approach, we have conducted a case study using a commercial requirements monitor and a collection of web service implementations available with Apache Axis. Our case study demonstrates the feasibility of verifying the conformance of a web service executing in an untrustworthy environment.

1 Introduction

With the growing popularity of web services, a lot of attention is being directed towards the specification and verification of functional and non-functional requirements of web services in a service oriented architecture paradigm. To monitor a service (or its composition) for functional requirements, such as "R1: the response given by the credit card verification service shall be *true* if the card number is valid and *false* otherwise", it is sufficient to observe or test the interface of the service. On the other hand, to validate a non-functional requirement such as "R2: the service shall not persist the credit card number supplied by the client", it may not be sufficient to validate just the external interface. There are a number of approaches such as by Barbon *et al.* [3] and Mahbub and Spanoudakis [11] to validate functional and non-functional requirements of a web service using dynamic monitoring.

Web services are often executed on servers that are not owned or operated by the clients. The validation for most non-functional requirements may only come from a monitor that is executing in the same domain as the service implementation and that can validate — by observing the running service implementation —that the requirements such as R2 are indeed satisfied. The design and development of these monitors is a widely studied problem in requirements monitoring literature (e.g. see [10, 12]). Nevertheless, the key question remains; *in a (possibly) untrustworthy domain who guarantees the integrity of the monitor? In other words, who monitors the monitor?*

The goal of our approach is as follows: given a set of service specification (S), a set of service implementation (I), a monitor that is capable of detecting deviations in the execution of the service implementation from its specification ($M : S \times I \rightarrow \{true, false\}$) running in a trusted environment, and a monitor that is similarly capable, but may be running in an untrustworthy environment ($M' : S \times I \rightarrow \{true, false\}$), how can we validate that $M \equiv M'$ is always true.

To give the reader an idea of the problem with verifying a monitor in an untrustworthy environment without a root of trust, let us for a moment assume that a validation mechanism $V' : M \times M' \rightarrow \{true, false\}$ exists. Now in order to answer this validation question, there must be a part of V' running in the same untrustworthy environment that can observe M' to compare it with M. If not, V' will depend on the untrustworthy environment to observe M', which in turn may mask the true responses of M' with expected responses for M thereby invalidating the premise that V' exists. On the other hand, if some part of V', say $\delta V'$ is running in the same untrustworthy environment to observe M', we will need another monitor to verify that the integrity of $\delta V'$ is not compromised, which will need to be verified again, *ad infinitum*. In summary, V' may not exist.

Using standards such as WS-Security [9] and WS-Trust [1] or proposals such as that by Skogsurd et al. [16], we can address the issue of security-token interoperability and secure messaging within SOA. But, these standards are not independently sufficient to guarantee indisputable trust in an untrustworthy environment. We propose to use a hardware-based mechanism as the root of trust for such validation mechanisms.

In the example described earlier, if we could be sure that

there exists a $\delta V'$ such that we do not need another monitor to verify its integrity, $\delta V'$ would make V' realizable. Fortunately, recent research results have shown that realization of such hardware-based root of trust is possible in the form of a *Trusted Platform Module* (TPM) [14, 13]. In this work, we describe an architecture based on TPM to validate the integrity of a runtime requirements monitor, which will in turn facilitate trusted services.

In [7], we studied the problem in detail and suggested a preliminary solution. It consisted of the TPM directly monitoring the web service implementation. This made the architecture inflexible with regards to bug-fixing and code evolution in the web service implementation. In the current proposal, we used a *requirements monitor* to monitor the integrity of the web service. The TPM monitors the integrity of this *requirements monitor*. Thus, the shortcomings of this architecture have been addressed.

Section 2 describes trusted platform modules, which form the basis of our proposed architecture. Section 3 discusses the approach for our new architecture. The experimental evaluation of a prototype implementation conforming to this architecture is discussed in Section 4. Section 5 compares and contrasts our work with related approaches. Section 6 discusses future work and concludes.

2 Background: Trusted Platform Module

A Trusted Platform Module (TPM) is a trusted agent coprocessor within a remote computing platform which derives its root of trust from its manufacturer or a delegated trusted third party [17]. A TPM can be trusted to perform certain actions truthfully despite being an integral part of a potentially malicious or compromised system. In other words, it is our trusted ambassador in a friendly or hostile foreign territory. A TPM provides roots of trust for storage, measurement, and reporting of measurement.

On every TPM, there is a facility for on-chip public and private key pair generation using the inbuilt hardware random number generator. This makes it possible for the TPM to encrypt and decrypt data. The TPM also has a set of registers called *Platform Configuration Registers*(PCR) which can be used to store the 160-bit hash values obtained using the SHA1 hashing algorithm of the TPM. The hardware ensures that the hash value of any PCR can be changed only by encrypting the new data over the previous hash value of the PCR. In this way, PCRs can be used to indelibly record the history of the machine since the last reboot. The PCRs are cleared off at the time of every reboot.

Over the past few years, the computer industry has come up with many initiatives to guarantee security, integrity and confidentiality of data through innovative hardware-based architectures. A consortium of key industry players, Trusted Computing Group (TCG) [17], came up with the specifications for a TPM with such a goal. The TCG vision was that this rudimentary TPM supported trust can be bootstrapped into a higher level trust through some software trust architecture or design principle. Hardware vendors are moving towards installing TPM on every computer that ships.

3 Approach

The main goal of this research is to preserve trust among the entities in a service oriented architecture. The current specification for web services in a service oriented architecture gives a lot of flexibility and freedom to the service providers and does not prevent them from implementing the services as they like. Currently, there is no fool-proof mechanism to verify the service provider's claim. Therefore, an approach is needed to guarantee the integrity of the web service. We employ the Clark-Wilson model to monitor the integrity of the requirements monitor which in turn monitors the web service.

Clark-Wilson's integrity model formalizes the concept of information integrity[4]. Clark-Wilson's model particularly emphasizes that the implementer of the transaction and the certifier of the transaction are essentially different entities. According to the model, any well-formed transaction should transition a system from one consistent state to another consistent state. To monitor this, there has to be a mechanism that transparently reports the state of the service provider's system from time to time. Such a mechanism should not be vulnerable to any kind of tampering.

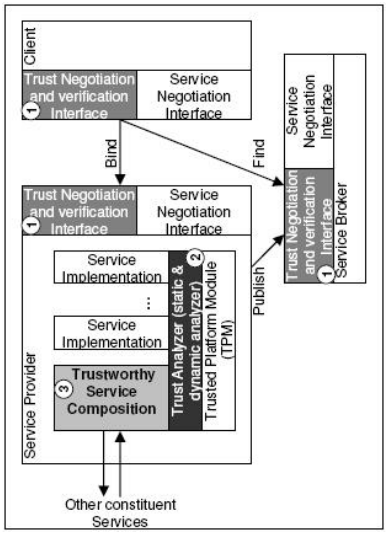

Figure 1. Our Proposed Architecture

Following the Clark-Wilson model, we proposed certain key additions to the standard SOA in the form of a new interface called *trust negotiation and verification interface* as shown in Figure 1 in [7]. The purpose of this interface was to provide an abstraction for the clients to negotiate desired integrity levels and for brokers to verify that the service im-

plementation was indeed conforming to the desired service specification. The trust negotiation and verification interface between the service broker and the service provider also allows broker to communicate with its trusted agent, the trusted platform module, and with service specific trust monitor in the service providers domain. The role of the trusted platform module is to periodically validate the integrity of the trust analyzer that in turn validates the conformance of the service implementation with the service specification.

We have implemented a system based on this hypothesized architecture to show the feasibility of our approach. Our system is shown in Figure 2. To recapitulate briefly, in a SOA there are three main entities: the service provider, the service broker and the client. The client contacts the service broker with a request and the broker directs the requester to the service provider. In our example system, the service broker also acts as the trusted third party. The monitor in this case can be any *requirements monitor* that verifies whether the service implementation on the service provider's side conforms to its quality requirements.

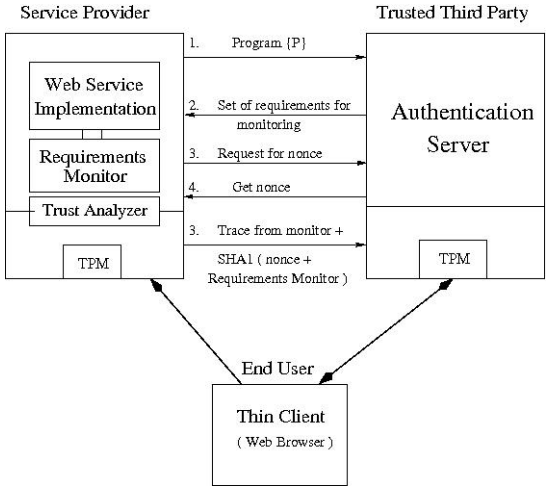

Figure 2. An Implementation of the Proposed Architecture

The trusted third party hosts an authentication server to authenticate whether the service implementation on the service provider's side is genuine. It does so by verifying execution trace of the service implementation using a *requirements monitor*. Since the scope for monitoring is very diverse, for demonstrating the feasibility of this system, we are assuming that if the service provider had malicious intentions, the service implementation would be modified to either store or process the confidential customer data. So, the goal of this architecture is to help the client to successfully complete the transaction with an assurance from the trusted third party that the service provider has not stored or processed the confidential data that were provided by the customer. In this architecture, we assume that the operating system on the service provider's environment is secure, implying that the service provider will not be able to change to monitoring software without knowledge of the TPM in the system. For example, the approach proposed by Sailer et al. to secure the operating system kernel can be used [13].

Initialization phase on service provider's side:

STEP 1. Accept the requirements to be monitored.

For every transaction, the following actions are carried out by the trust analyzer:

STEP 1. Accept nonce from the authentication server on the trusted third party.

STEP 2. Generate a trace for the specific web service implementation program using the *requirements monitor* and send it to trusted third party.

STEP 3. Using the local TPM, compute the SHA1 hash of the software stack up to the *requirements monitor* including the nonce.

STEP 4. Encrypt the hash using the public part of AIK of the TPM of the trusted third party and send the encrypted data to it.

Figure 3. Procedure for establishing trust in the service provider's environment

The algorithm for verifying the integrity is described in Figure 3, Figure 4 and Figure 5. The initialization phase is carried out under the supervision of a trusted authority. During this phase, the authentication server identifies the requirements to be monitored. We emulated the requirements identification process which consists of determining variables and methods dealing with data labeled as sensitive, by using Kaveri [8], a tool for program slicing. In future, we plan to have the requirement specifications as a part of the web service interface itself, thereby decoupling the process of requirements identification, which is currently tied to the implementation of the web service.

For every transaction, the authentication server generates a nonce to guard against replay attacks. A nonce is a random number that is generated only once and is included in all interactions to prove the freshness of data. It is to be noted that we are not checking the trace for an exact equivalence. This will lead to false positives because a program can be modified for purposes such as bug-fixing, without violating the specifications. Therefore, we programmatically check for violation of specific properties listed before. If any of the system, configuration or library files up to the *requirements monitor* are even slightly tampered, there will be significant variations in the final SHA1 hash value. It takes about 2^{69} units of time to find SHA1 collisions ac-

Initialization phase on the trusted third party:

STEP 1. Identify the requirements to be monitored.

STEP 2. Install the *requirements monitor* and on the program and generate a trace for the requirements determined above.

STEP 3. Using TPM, measure the software stack up to the *requirements monitor* and store this measurement for future reference.

For every transaction, the following are done by the authentication server:

STEP 1. Send a nonce to the service provider.

STEP 2. Receive the trace from the *requirements monitor* for the specific execution of the web service.

STEP 3. Receive the encrypted data from the service provider.

STEP 4. Decrypt the data in the TPM using the private part of it's non-migratable AIK to get the TPM measurement of the service provider's *requirements monitor* and verify the value for conformance.

STEP 5. With the reference measurement as the basis, programmatically check the trace for violation of specific properties. Check for replay attacks.

STEP 6. If these checks fail, notify the user of a possibility of violation.

Figure 4. Procedure for monitoring trust from the trusted third party's environment

On the client's side:

STEP 1. Notify the trusted third party before the transaction.

STEP 2. Send request to the web service.

STEP 3. Wait for an assurance/notification from the trusted third party.

Figure 5. Procedure to be followed by the client for conducting a transaction

cording [19], implying that collisions are very rare. Hence, these variations can be detected easily.

In step 4 of Figure 3, AIK (Attestation Identity Key) is used to encrypt data to make the interactions secure. AIK is a special purpose asymmetric signature key created by the TPM manufacturer, the private portion of which is non-migratable and protected by the TPM. Since the private part of the TPM's AIK cannot be retrieved by any user, the decryption of the data has to be done only on the trusted third party's local machine using the private part of it's AIK.

4 Case Study

The subjects for our case study were selected from the sample web service implementations available from the Apache Axis distribution. Table 6 briefly describes the web services used for this case study and the sections of the service implementation that was traced by the requirement monitor for each web service. Some of these sections were chosen randomly while others were chosen to monitor certain methods handling specific data, labeled as sensitive.

Service name	Short description	Traced Sections of the Service Implementation
Stock	Gets quote for the stock "symbol"	1. Instructions invoking setters. 2. Methods with private access.
Echo	Echoes a string	1. Method entries and exits. 2. Methods with public access.
Encoding	Serialization of a message	1. Methods with private access. 2. Instructions which invoke getters.
Message	Simple XML messaging service	1. Methods with private access. 2. Instructions invoking getters.
Bidbuy	Request for a quote, purchase a given quantity of a specified product and process purchase order.	1. Method entries and exits. 2. Instructions that invoke getters and setters.

Figure 6. Subjects for our Case Study

4.1 Experimental Setup

The experimental setup was implemented using two Dell Precision 390 stations each having Intel Core2 Duo Processor @ 1.86 GHz and 2 GB of RAM. The processors on these stations have a TPM (Version 1.2) manufactured by Atmel Corporation, embedded in them with 24 PCRs each. One of the stations is assigned the role of a service provider while the other plays the role of a trusted third party. We used *tpm4java* for developing our trust analyzer to take integrity measurements of the requirements monitor on the service provider's side. The Java library *tpm4java*, developed by Tews et al. [18], is used for accessing the TPM functionality from Java applications. The test environment consists of Apache Web server Version 2.2, Tomcat Servlet Container Version 5.5.23 and Axis SOAP server Version 1.2 running on Windows XP Professional operating system. For evaluating the requirements of the web service implementation, we use a commercial software called $CodeMonitor^{TM}$ (monitor) from Tangentum Technologies [5] as our subject monitor. The monitor instruments the Java bytecode to log certain actions and this makes it possible to monitor web services that have already been deployed. For doing these, it must be installed in the same environment as that of the web service. For the purpose of this experiment, we defined the requirement as, *" The execution trace of the program involving the variables and methods dealing with client data labeled as sensitive, should not include APIs dealing with persistence or serialization."*

4.2 Violations

This class of compromise can be detected by current approaches for requirements monitoring (e.g. [10, 12]). Using similar techniques, our subject monitor was also able to give the execution trace for methods that caused either persistence or serialization of data. When such a violation occurs, the trusted third party can signal the end user of a breach of trust by the web service.

Since the monitor has to be installed in the service provider's environment, the monitor itself can be compromised in many ways. For this paper, we instrumented the monitor to report a normal trace even when there was a violation of trust. Thus, the integrity of the web service is a function of the integrity of the requirements monitoring software. One such case is presented in Figure 7, in which one of the library files of the monitor is altered. Since the monitor itself is being monitored by the hardware-based TPM, it is possible to detect such a violation.

Figure 7. TPM Measurements for a Genuine and a Compromised Requirements Monitor

Files Monitored by TPM	160-bit SHA1 Hash of Genuine Monitor	160-bit SHA1 of the Compromised Monitor
../codemonitor.license	6476...DB8F	6476...DB8F
	...	
../codemonitor.config	8D9E...5FA8	8D9E...5FA8
../codemonitor.jar	23F9...5BA2	D843...1531
../jbcs-client.jar	86AA...BE56	0F66...00F7
	...	

From Figure 7, it can be observed that the hash values in the third column starting from the entry corresponding to the file *codemonitor.jar* differ significantly from their corresponding entries in the second column. This is because the SHA1 hashing algorithm in the TPM not only hashes the listed files but also preserves the order of hashing. It implies that at least one of the library files including *codemonitor.jar* has been altered without the knowledge of the trusted third party.

4.3 Overhead of Monitoring

Table 1 compares the average time taken to execute a web service in a standalone manner, when CodeMonitor is used and when custom Aspects are applied for monitoring the web service implementation. These values are the averages of the time taken to execute the service over ten client requests. The overhead due to CodeMonitor is greater because, it instruments all the instructions used in the web service implementation including those of the libraries, at run time. Since the source code for CodeMonitor was not available, we could not circumvent this overhead. So, we wrote custom aspects to monitor the same sections of the service implementation and achieved a better performance. This demonstrates that web services can be monitored for integrity without a tangible time lag in responding to the client's request.

Table 1. Overhead of Monitoring

Service	Execution Time without any monitor (in seconds)	Execution time with *CodeMonitor* (in seconds)	Execution time with Aspect Monitor
Stock	0.944	10.688 11.476	1.283 1.005
Echo	1.299	42.375 12.812	1.609 1.640
Encoding	0.738	11.828 9.621	0.922 1.026
Message	0.945	7.200 20.641	1.209 1.208
Bidbuy	0.993	83.110 10.900	1.349 1.341

5 Related Work

Ever since the 1970s, efforts have been made to produce secure operating systems [15] as a basis for secure computing. In 1997, Arabaugh et al. proposed an architecture for secure and reliable bootstrapping called AEGIS [2].

In 2003, Grafinkel et al. proposed Terra [6], a virtual-machine based platform for trusted computing. In Terra, a virtual machine monitor was used to simultaneously partition the hardware into independent, isolated virtual machines of varying levels of security. Unlike AEGIS, Terra does not start from a secure boot process.

In 2004, Sailer et al. proposed a TCG based Integrity Measurement Architecture for Linux [14]. This architecture made use of a TPM hardware to store the integrity measurements of the system. The purpose of this architecture is to present an ordered list of measurements to a remote party. The remote system determines the integrity of the attested system by reconstructing the image of the attested system's software stack on the local system using these measurements and then by applying the security policy on the local software stack. To establish mutual trust, this process has to be carried out on both sides involved in the transaction [13]. This was implemented by instrumenting the Linux kernel to create measurements and to store them in the TPM hardware to protect against compromise.

However, the process of recreating the image of the other party on the local system based on the measurements obtained is complex. The task of taking measurements is implemented hacking the Linux kernel code. In case of online transactions, common users may not have the Linux operating system. In a majority of the cases, the two communicating parties may not have the same operating system in

their environments. Our architecture is designed to address these issues by delegating the task of certifying the service provider to a trusted third party.

6 Conclusion and Future Work

Existing security frameworks do not offer any guarantee to the client whether the data will remain private and tamper-proof in the domain of the service provider. In a truly decoupled environment, which is the main motto of SOAs, including constructs to negotiate, enforce, and verify trust and security guarantees within the provider's domain through the service discovery interfaces thus seems to be a crucial pre-condition for mission-critical deployment of SOAs. Our proposed architecture for ensuring the integrity of requirement monitors is a step in this direction. Our experimental results demonstrate the viability of monitoring web services for integrity without incurring a tangible overhead in terms of time. In future, the architecture will undergo major enhancements to include a framework for the client to specify certain properties that can be enforced on the web service implementation by the requirements monitor.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grants 0540362 and 0627354.

References

[1] S. Anderson and et al. Web services trust language (wstrust). http://msdn.microsoft.com/ws/2004/04/ws-trust/.

[2] W. A. Arbaugh, D. J. farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symp. Security and Privacy*, pages 65–71, 1997.

[3] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06*, pages 63–71.

[4] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[5] $codemonitor^{TM}$. http://www.tangentum.biz/.

[6] T. GARFINKEL, B. PFAFF, J. CHOW, M. ROSENBLUM, and D. BONEH. Terra: A virtual machine-based platform for trusted computing, 2003.

[7] M. Hosamani, H. Narayanappa, and H. Rajan. Monitoring the monitor: An approach towards trustworthiness in service oriented architecture. In *2nd International Workshop on Service Oriented Software Engineering*, September 2007.

[8] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering indus java program slicer. In *Fundamental Approaches to Software Engineering, FASE 2005, Springer-Verlag*, April 2005.

[9] C. Kaler and et al. Web services security (ws-security). http://msdn.microsoft.com/library/enus/dnglobspec/html/ws-security.asp.

[10] E. Letier, J. Kramer, J. Magee, and S. Uchitel. Monitoring and control in scenario-based requirements analysis. In *ICSE '05*, pages 382–391.

[11] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. In *ICWS '05*, pages 257–265.

[12] W. Robinson. Monitoring software requirements using instrumented code. In *HICSS '02*, page 276.2.

[13] R. Sailer, L. van Doorn, and J. P. Ward. The role of TPM in enterprise security. Technical report, IBM Research, October 2004.

[14] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Thirteenth Usenix Security Symposium*, pages 223–238, August 2004.

[15] M. Schroeder. Engineering a security kernel for multics. In *Fifth Symposium on Operating Systems Principles*, pages 125–132, November 1975.

[16] H. Skogsrud, B. Benatallah, F. Casati, and F. Toumani. Managing impacts of security protocol changes in service-oriented applications. In *2007 IEEE International Conference on Software Engineering*, 2007.

[17] Trusted computing group. https://www.trustedcomputinggroup.org.

[18] E. Tews and M. Hermanowski. Projektvorstellung tpm4java trusted computing fur java. http://tpm4java.datenzone.de.

[19] X. Wang, Y. L. Yin, and H. Yu. Collision search attacks on sha1. http://www.cryptome.org/sha-attacks.htm.