xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement*

Gabriela Gheorghe, Stephan Neuhaus, and Bruno Crispo

Università degli Studi di Trento, I-38100 Trento, Italy
First.Last@disi.unitn.it

Abstract. Enforcing complex policies that span organizational domains is an open challenge. Current work on SOA policy enforcement splits security in logical components that can be distributed across domains, but does not offer any concrete solution to *integrate* this security functionality so that it works across security services for organization-wide policies. In this paper, we propose xESB, an enhanced version of an Enterprise Message Bus (ESB), where we monitor and enforce preventive and reactive policies, both for access control and usage control policies, and both inside one domain and between domains. In addition, we introduce indicators that help SOA administrators assess the effectiveness of their policies. Our performance measurements show that policy enforcement at the ESB level comes with only moderate penalties.

1 Introduction

As Service-oriented architectures (SOAs) expand, they need to interconnect and adapt to increasing business and infrastructural demands. These intercommunication and interconnection requirements are met by a piece of middleware called the Enterprise Service Bus (ESB). The ESB offers a standard way to connect services by acting as a message hub, making interservice communication smooth and painless. But this ease of use comes with a downside: when businesses expose their services in order to participate in an SOA, they face increased risks of misuse or abuse. What is therefore needed is a way to formulate and enforce policies that make such misuse or abuse impossible.

However, ESBs do not address non-functional aspects such as security, so what protects messages in transit and makes security decisions based on them? Since ESB components are using the bus as a low-level service which abstracts communication details from the higher-level services, we believe that it is the ESB that should be in charge of enforcing message-level policies.

There are two main aspects in which current work in SOA policy enforcement falls short of this expectation. First, the majority of existing work locates policy enforcement inside an orchestration engine such as BPEL, but such a view is not suited to scenarios where policies concern the service request or response

* This work is supported by the European Commission under the project EU-FP7-IST-IP-MASTER.

messages themselves instead of the effect they have on the business process. Second, current approaches focus mostly on simple access control policies instead of complex organization-wide policies that also include usage control.

Another problem, one that plagues administrators of SOAs, is whether security policies perform as they should. The state of the art is to pepper the deployed services with debugging output, but this is clearly unsatisfactory, if not completely infeasible, for example when using third-party services.

This paper addresses all three issues by implementing flexible, instrumentable, and highly configurable policy enforcement mechanisms at the ESB level. We also cover access as well as usage control policies that can span the entire organization: our enforcement mechanisms implement *reactions* to violations in addition to traditional access control. Additionally, we augment policies with *indicators* that our ESB continuously monitors in order to measure how well the policies perform. By providing a unified and service-independent way to aggregate data, indicators help administrators understand, at run-time, why policies fail.

Besides addressing the issues of regulating message flow and allowing organisation-wide policies, our work also decouples the enforcement logic from the business logic. This way, process and application designers can focus on the business aspects of their applications and how they must be used, but not how this is technically enforced. Thus our contributions can be summarized as follows:

– A *new approach* for security policy enforcement for SOA (Sect. 4).
– A *working prototype implementation of* xESB, (Sect. 5), that has *low overhead* (Sect. 8).
– Support for *reactive policies*, and for *usage control policies* (Sect. 6).
– Support for *indicators* to help monitoring policies (Sect. 7).

The remainder of this paper is organized as follows. After showing our supporting case study (Sect. 2) and some background on the ESB platform (Sect. 3), we motivate why policy enforcement on the ESB is needed (Sect. 4). After that, we introduce our xESB prototype (Sect. 5) and briefly present the language we used to write our policies (Sect. 6). We next give an overview of some possible enforcement indicators (Sect. 7), and present a performance evaluation (Sect. 8). We finish with a review of related work (Sect. 9) and a discussion of future work and conclusions (Sect. 10).

2 Motivating Example

As motivating example to illustrate the design and features of xESB, we consider a hypothetical company 'Foo.uk', providing VoIP-based services using a communication platform implemented as a SOA using an ESB. While Foo.uk is hypothetical, the problems it faces definitely are not. For example, Zimmermann et al. published a large case study in which a "large telecommunication wholesaler" switches to BPEL and SOA, so it is clear that the wholesaler will also have to face security issues at the SOA level [1]. However, giving examples from actual companies would give much extraneous detail not relevant to this paper, and we have therefore stripped the examples down so that the relevant problems and their solutions can be exposed more clearly.

Since Foo.uk operates in the UK, it must comply with regulations such as those described in the Statutory Instrument 2003 No. 2426 [2] implementing the Privacy and Electronic Communications EC Directive [3]. In order to do so, the Foo.uk platform must be able to enforce policies such as "Log starting time and duration of incoming calls" or "Hide initiator number in outgoing calls".

For the first policy, a simple mechanism should signal the start of a call and its duration. Capturing and logging these events should leave the application unaltered. What is therefore required is a control that is able to *filter* messages and to *duplicate* them to a logging service. For the second policy, a mechanism is needed to *filter* outgoing calls from incoming ones. This can be achieved simply by looking at the type and parameters of the event (either a service invocation or a service response) and then *modifying* those that identify the initiator. In addition, for business purposes Foo.uk must be able to enforce also policies like:

Silver customers can use premium services only for 3 hours a month. If the control service assigns special message identification that can easily differentiate between subscriber types and call types, Foo.uk needs logic to infer the duration of a call for a specified amount of time. Anytime a silver customer would request a video call, the request should not be serviced unless the 3 hours a month have not been exceeded. Thus, the service response depends on whether a predicate is satisfied or not.

Process collect calls only after destination has agreed to pay. Similar to the previous case, verifying a condition in this case means ensuring that something has happened in the past: the VoIP destination must have accepted to pay for the call. This predicate needs to be checked only before replying to a collect call request: the response is delayed until the destination either explicitly accepts or rejects the payment or times out (which should be construed as rejection).

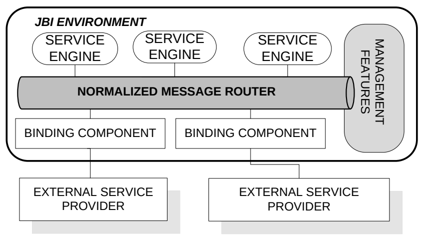
Delay high-quality calls until resources are available. Assuming a VoIP user requests high quality parameters for a call, a load-balancer component of the SOA would have the authority to *delay* the service request until it allocates the resources needed for such a situation.

Enacting the rules above can be done by a dedicated component acting as gateway for rule compliance: as an infrastructure component, it would interpose between the VoIP provider and the service clients. Having an application-level module in charge of this would be inefficient because the above policies do not directly relate to application logic; they concern legal issues that the VoIP communication protocol should obey, irrespective of its conceptual design or its architecture. These issues, or constraints, can be more frequently subject to change (or update), than the overall SOA application. We argue that it is best to separate the constraint checking functionality from the business application in such a way that the former can easily *adapt to new organizational requirements*. If a new business or regulatory policy would replace an existing one (e.g., not hide, but *encrypt* initiator number in outgoing calls), then the changes on the enforcement logic should be as light as changing a service endpoint (from one that hides data to one that encrypts data). This behavior would be in complete resonance with the concepts of service-orientation and reuse, because it brings a clear decoupling between service logic and service security. Such a separation has not been previously addressed at the message level, and our solution benefits from this approach in that the prototype is not *hardcoded* to a specific decision making or enforcement component; any trusted security components of these types can be plugged in.

3 The ESB in the Service-Oriented Architecture

The ESB is a middleware placed between the various services of an SOA application. It offers a layer of communication and integration logic in order to mitigate technology disparities between communication parties, ranging from intelligent routing to XML data transformation.

Java Business Integration (JBI) [4] standardizes deployment and management of services deployed on an ESB. It describes how applications need to be built in order to be integrated easily. The generic JBI architecture is shown in Figure 1 (left). Since the primary function of the ESB is message mediation, the Normalized Message Router (NMR) is the core of the JBI-ESB and provides the infrastructure for all message exchanges once they are transformed into a normalized form. Components deployed onto the bus can be either service engines, which encapsulate business logic and transformation services; or binding components, which connect external services to the JBI environment.



```
InOnly[
 id: ID:192.168.233.83-1228375ebcb-8:0
 status: Active
 role: provider
 service: {http://www.microsoft.com}acceptReceiver1
 endpoint: acceptor1
 in: <?xml version=1.0" encoding="’UTF-8"?><example
       id="123"/> ]
```

Fig. 1. The architecture of the JBI system (left) and example of a normalized message in an InOnly exchange (right)

4 The Enforcement Process

The runtime enforcement process starts the moment a message is intercepted. Once obtained, the event is evaluated against the deployed security policies; the result is a decision that the enforcer translates into a series of actions; the enforcement process is in charge of performing those actions (either directly or by delegating them to a trusted third party). This section describes how we modeled the policy enforcement process that we later implemented in xESB.

The policy language and its interpretation are the subject of Section 6.

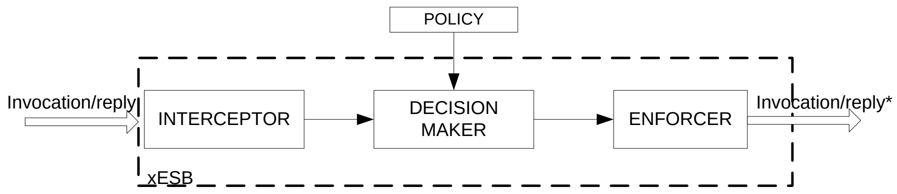


Fig. 2. The enforcement process behind xESB

4.1 Interception

Enforcement starts by intercepting messages to which at least one policy applies. The policies dictate what message types and parameters to look at: message destination, source, size, or metadata like annotation information. Irrespective of the message format (usually XML-based), the elements described above are usually easy to inspect on every message simply because all messages on the bus have the same format. A prefiltering mechanism can help the interceptor catch messages with a higher probability of being relevant than any other message; for instance, if a policy refers to requests that are simultaneously outgoing *and* have a valid security signature, then the interceptor could just check if the current call is outgoing. This is a condition which is inexpensive to check compared to the validity of the signature, and if it does not hold then the signature need not be validated. Figure 1 (right) shows the format of a message on the NMR; because of the normalized format, split into structured metadata and payload, the message destination and the direction of the message can be easily extracted and compared against given data.

This simple mechanism is an easy way to separate incoming calls from outgoing ones, based on message metadata.

4.2 Decision

Once a policy-relevant message is intercepted, it needs to be evaluated against the applicable policy (see Fig.2). The decision component does policy matching: it examines all policies in the policy base (or policy repository) by evaluating them against the current message. The evaluation is done by comparing message context and actual parameters (e.g., destination service, source service, message type, etc) with the conditions required by interested policies. We considered the simple case of comparing the message against all policies in the policy base, until the first match is found.

The output of the decision phase is called the *verdict*. This enforcement decision is binary: either the message is legitimate, or it is a policy violation. While the first case implies that there will be no consequences onto the message flow, the second case calls for one or more enforcement actions. These actions are detailed below.

4.3 Actions

The third step of enforcement is to take action based on the verdict that was reached in step 2. We have implemented five basic enforcement actions that implement the four mechanisms formalized by Pretschner et al. [5] to approach usage control enforcement:

The acceptor accepts whole messages. If the verdict does not indicate any policy violation, then the acceptor is invoked, with the effect that the message is allowed without any modification.

The blocker rejects whole messages. Contrary to the previous case, the blocker mechanism is invoked to react to a policy violation by rejecting the entire message and sending back an error message.

The modifier pertains to the class of mechanisms that modify a message. The point is to go beyond the classic “all or nothing” enforcement approach and modify the message so that it conforms to the given policy.

The delayer refers to the class of mechanisms that postpone a message until a condition is satisfied. This mechanism maps to the idea of obligation enforcement, where an actor would be allowed to perform an action only after a condition has been verified. For instance the policy “Delete all traces of a call after the call has been terminated and paid” can be implemented by delaying the deletion of call traces until the arrival of a message that signals a call that is terminated and paid for.

The executor enables complex actions. In some cases, the reaction to a violation may require the execution of a complex recovery process that requires more than the basic mechanisms implemented by xESB. Implementing these mechanisms in xESB would make it inflexible, so xESB uses the executor to invoke an external service or process, which can also be an orchestration engine.

Actions can be differentiated as *preventive* or *corrective*. Preventive actions ensure that a policy violation will not happen: prior to allowing a sensitive action, they check its compliance with the policies. Corrective actions, on the other hand, try to compensate a violation that already happened. The blocker is a preventive mechanism: if a message on the ESB is not allowed to reach its destination, then it is simply dropped. The other actions—modifying, delaying, calling an external entity—are by their nature compensatory: if an intercepted message or a group of messages already constitute a policy violation, an appropriate action must be taken to *correct* the respective message or message flow (e.g., reroute to a secure service, deny further messages on that route until next day, etc.). The executor mechanism can be both preventive and reactive.

We see the actions described above as *enforcement primitives* on ESB messaging. Because implementing a blocking, modifying, delaying and executing mechanism should be different from application to application, we argue that our design caters to a *customizable* ESB enforcement framework. Our model provides a set of basic components – the interceptor, the decision maker, the action performer – and the wiring between them; the semantic behind the enforcement actions and the policies are independent of the solution design.

5 The Design of xESB

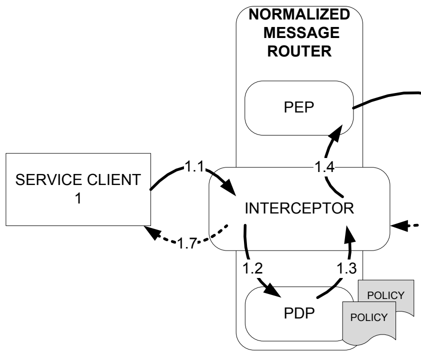


Fig. 3. The xESB enforcement architecture. The solid arrows indicate the invoke chain, while the dotted arrows show the response chain. Both request and response are intercepted. The policy is only on requests from Client 1 to Provider 2, hence the interceptor gives the request to the PDP, and then passes the response from the PDP to the PEP.

In a JBI service bus, the component that mediates all communication is the normalized message router (NMR). It is therefore natural to embed an *interposition mechanism* within the NMR, in order to capture incoming messages on their way to their destination, be it before being processed (service requests) or after being processed (service replies).

Once the message has been intercepted, the next step is to *analyze* the messages just captured. The JBI standard helps us again: the messages being routed are normalized before they get to the router; this means that they are transformed to a fixed format that separates clearly the XML payload of a message, the message context (metadata) and message attachments (the format is given in Figure 1). This feature makes it feasible to analyze parts of a normalized message. In order to derive a verdict, we designed the analysis component to compare the metadata of a current message with the metadata specified by the policies in the policy base. By metadata, we mean information such as message source and destination; by context we mean any flow-related condition, such as a constraint that pertains to message flow precedence (e.g., an incoming call was not logged before it was accepted), or usage control-specific constraints that mainly involve counting (e.g., a silver customer has used 2 hours of premium services this month) or obligations (e.g., silver customers can use video calls for no more than 3 hours a month). With some information on the state of a service or a message, the problem of comparing the current intercepted message against the business constraints in the policy base is described in Sect. 6.

Once a verdict is reached, the *action performing* phase consists of one of the actions given in Sec. 4.3. xESB implements message blocking by sending the current message not to its desired endpoint but to a loopback interface (the request initiator can receive nothing or a fault message); the xESB modifier simply replaces parts of the message header as specified by the policy; the delayer routes the message to a delayer component that puts the message in a queue and removes it again when the condition associated with it is satisfied (be it time or a boolean condition); lastly, the executor is implemented by routing the current message to the endpoint of the desired service. An underlying assumption of the design is that the xESB mechanism is always invoked because the NMR processes every message. The xESB is trusted to always invoke the PEP, and the enforcement actions are on services that are deployed on the ESB (hence a service not deployed on the bus is not known to xESB). Therefore, xESB acts as a reference monitor.

Performance considerations made us embed the modifying and rerouting functionality into the component that does message interception on the bus. The specific delaying and blocking behavior were implemented as separate components (namely, plain Java objects) to which the NMR would direct messages. Thus, while the intercepting mechanism is ESB platform-specific, we aimed to make the analysis and action hooks reusable across applications. The logic to decide on the enforcement verdict can be reused irrespective of the deployed policies, but the actions matching this verdict are application-dependent. This design has *flexibility* as its greatest advantage; for instance, it does not rule out the call-back to an analysis component that is application-logic aware, and that might be interested to analyze message payload in order to make an enforcement decision. In addition to this aspect, we have also incorporated support for indicators within our xESB. Section 7 will discuss indicators in greater detail.

6 Enforcement Language

As we have shown in Sect. 4.3, we wish to enforce not only by control, but also by reaction. Reaction covers temporal and transient obligations to which any entity in the distributed environment can be bound. Existing languages and language frameworks address only some of these aspects (see Sect. 9). Consequently, we have created our own policy language.

A policy is written as text, which is compiled into binary form. During execution, the compiled policy file resides in memory and is interpreted by a stack machine. Policy turnover is at the moment not implemented, but we do not see large technical difficulties in doing so: turnover is problematic only for enforcement on message *exchanges* or *sessions*, where different policies may have to be applied to different messages. Since we enforce policies one message at a time, such problems do not exist in xESB.

6.1 Policy Files

Policies are expressed as a sequences of rules, expressed in Event-Condition-Action form. A message is sequentially checked against the rules until either a rule applies or there are no more rules. Figure 4 shows how the policy "Silver customers can use video calls for no more than 3 hours a month" would be expressed in this language. From this example, we can see that the policy language contains the following components:

```
default-action { allow; }
// Total duration of video calls, in seconds
hash videoDuration = 0;
timer resetDuration = next month;
obligation {
  if invocation
  when { resetDuration.fired }
  do {
    clear videoDuration;
    arm resetDuration fire next month;
  }
}
obligation {
  if response
  when { h "Type" equals "video-call" && h "Success" equals "True"
    && h "Customer-Type" equals "Silver" }
  do { update-counter videoDuration[source] += h "Duration"; }
}
rule {
  if invocation
  when { h "Type" equals "video-call" && h "Customer-Type" equals "Silver"
    && videoDuration[source] > 10800 }
  do { block; }
}
```

Fig. 4. Policy for "Silver customers can use video calls for at most 3 hours a month".

Default Action. This allows *allow-based* or *deny-based* policies.

Counters, Timers, and Hashes. These declare items of state, which are pieces of data that keep their values across policy checks. There are three types of state: *counters*, *timers*, and *hashes*. The latter keep arrays of state, thus allowing state per user or state per messge source etc.

Rules and Obligations. Rules and obligations are very similar. However, *rules* compute verdicts such as `block`, `allow` and so on. When a rule that carries a verdict matches a message, the action part of that rule is executed and processing is stopped. On the other hand, *obligations* exist solely for the purpose of updating state, so processing continues.

Event, Condition and Action Specifications. The event part of a rule or obligation checks if the message is a *request* or a *response*. Conditions are part of a rule or obligation and *checks whether the rule or obligation applies* to the current message. The action specification of a rule or obligation can *update state* (both rules and obligations) or *return a verdict* (rules only).

What is the overall effect of this policy file? The first obligation takes care of rearming the timer if it has fired. The second obligation updates the length of video calls, and the rule blocks video calls in excess of three hours. Identifiers such as 'type', 'source', 'destination', etc. refer to names of the metadata fields in the normalized message.

While this example illustrates the main features of the language, some other features of interest include:

Modifying message metadata. The language construct "`modify h` *metadata-name* `=` *string-expression*" modifies parts of a message's metadata, i.e., anything outside the message payload. Modification lets the message pass after modification.

Delaying a message. One possible verdict is "`delay` *n*", which means to delay the message by a specified amount of time. This implies allowing the message to pass eventually.

Delay until a condition is met. Another innovative verdict is "`delay until` *condition*", which will delay a message until a certain condition is met. The condition can be any boolean expression on the state (but not on any message headers). This is not the same as bocking a message, since a message is completely *discarded* when it is blocked, whereas here it is merely *delayed*.

6.2 Cross-Service Policies

To show that we can use xESB to enforce cross-service and hence potentially inter-organisational policies, let us consider the regulatory requirement to hide initiator numbers in outgoing calls. Since this policy holds for video, audio and ordinary phone calls, it affects potentially many services and therefore also potentially different organisations within Foo.uk. Figure 5 shows how to express this policy. Note how simple this policy is to implement on the ESB level. On the BPEL level, it would be much more complicated, because a generic anonymisation service would have to be written and deployed, and message transformation would have to be performed at the BPEL level.

```
default-action { allow; }
rule {
  if invocation
  when { h "Type" equals "start-call" }
  do { modify h "Initiator" = "000000"; }
}
```

Fig. 5. Policy expressing "Initiator numbers need to be anonymized for outgoing calls".[2]

Figure 6 shows the two remaining policies from Sect. 2, namely "log start time and duration of calls", and "accept collect calls only after destination explicitly accepts to pay".

```
default-action { allow; }
rule {
  if invocation
  when { h "Type" equals "start-call"
      || h "Type" equals "end-call" }
  do { duplicate
    "http://internal.foo.uk/log"; }
}
```

```
default-action { allow; }
hash payAccepted = 0;
obligation {
  if request
  when { h "Type" equals "collect-payment" }
  do { update-counter payAccepted[destination] := 0; }
}
obligation {
  if response
  when { h "Type" equals "collect-payment"
      && h "Success" equals "True"
      && h "PaymentAgreed" equals "True" }
  do { update-counter payAccepted[destination] := 1; }
}
rule {
  if invocation
  when { h "Type" equals "call-collect"
      && payAccepted[destination] == 0 }
  do { block; }
}
```

Fig. 6. Policies expressing "log start time and duration of calls" (left), and "accept collect calls only after destination accepts to pay" (right).

7 Enforcement Indicators

As previously mentioned, we want indicators to give a quantitative measure of the quality of the enforcement process. We will provide some examples of possible ESB-level indicators that have an impact over the assessment of policy enforcement. We split them in two basic types:

Indicators for misconfiguration. By counting repeated violations from a particular service, an indicator can show that the service always violates the policy no matter the user on whose behalf it works. This would mean that the cause is not the user nor the way the service is used, but rather the way in which the service is configured. Another example is an indicator that can quantify how many services do not follow deployment or runtime constraints such as: using disallowed protocol versions, being in disallowed service states or deploying services that should not have been deployed. That may be a more general indicator for misconfiguration of the overall application. We show two simple examples in Fig. 7.

Indicators for reaction to misuse/attacks. Recursion in service chains can impact service availability because it can lead to deadlocks. Counting and limiting the number of times this happens may be an indicator of service availability. An example of an indicator preventing attacks is disabling access to

[2] The `modify` action implies a verdict, hence this is indeed a rule, not an obligation.

```
counter violations = 0;
rule {
  if request
  when { ... }
  do {
    update-counter violations += 1;
    block;
  }
}
```

```
counter violations = 0;
obligation {
  if request
  when { h "Protocol-Version" != "1.3"
      || h "Service-Type" != "ShoppingCart" }
  do { update-counter violations += 1; }
}
```

Fig. 7. Indicators for misconfiguration: Counting repeated violations (left), counting deployment errors (right)

```
counter recursion = 0;
counter aCalledB = 0;
obligation {
  if request
  when { source = "a"
      && destination != "b" }
  do { update-counter aCalledB := 0; } }
obligation {
  if request
  when { source = "a"
      && destination = "b" }
  do { update-counter aCalledB := 1; } }
obligation {
  if request
  when { source = "b"
      && destination = "a" && aCalledB }
  do { update-counter recursion += 1; } }
```

```
hash fails = 0;
obligation {
  if response
  when { source = "Payment" && h "Status" = "Fail" }
  do { update-counter fails[destination] += 1; } }
rule {
  if response
  when { source = "Payment" && h "Status" = "Success" }
  do { update-counter fails[destination] := 0; } }
rule {
  if response
  when { source = "Fail-Admin" && h "Status" = "Success" }
  do { pass;
       update-counter fails[h "Subject"] := 0; } }
rule {
  if request
  when { destination = "Payment" && fails[source] > 3 }
  do { block; } }
```

Fig. 8. Indicators for misuse or attack: counting recursions (left), preventing DoS or password guessing (right).

services based on security parameters or on the history of requests from a caller. For example, if a caller is denied access three times in a row, we block access permanently. Limiting the number of requests from a particular client to a particular service can prevent DoS attacks; see Fig. 8.

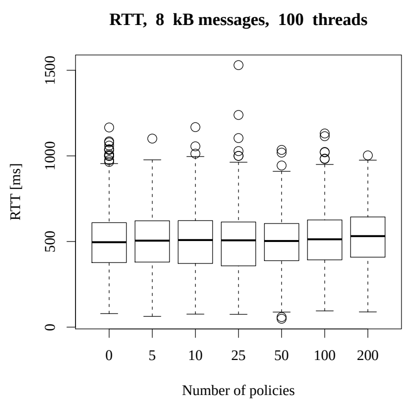
It can be noticed from the examples above that we can derive a number of useful indicators by combining mechanisms for counting, using flags for flow precedence, inspecting message types, sizes and message parts. Our language and our intercepting mechanism can be used to count specific events and reason on message precedence, as well as inspecting message properties.

8 Performance Evaluation

To implement xESB, we chose Apache Servicemix 3.3, a JBI-compliant open source ESB. We used the Servicemix API to intercept messages according to Fig. 3. This section describes the evaluation of our prototype implementation.

We followed a capacity testing model [6] to measure the lower bound of the instrumentation overhead. We used the sample SOAP that come with ServiceMix 3.3, and soapUI as a tool for load generation[3]. We used SOAP messages of 8 Kb

[3] http://www.soapui.org/



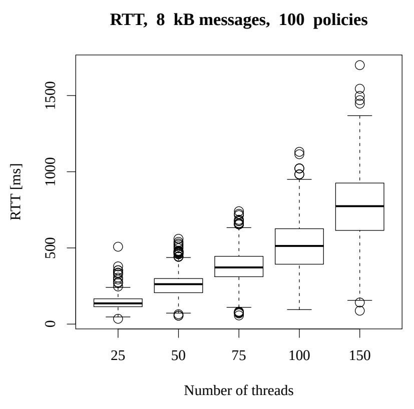


Fig. 9. RTT for varying policy file sizes (left) and varying number of parallel connections (right), for 8 Kb messages. The x axes are not uniformly spaced.

size and a varying number of parallel clients. Our testbed PC was a 32-bit system with a 2.6GHz processor and 3GB of RAM. The JVM was allowed 1280MB of memory, with a ServiceMix queue size of 256 requests.

To answer the question "how does the number of rules in a policy file affect the round-trip time (RTT) of messages?", we constructed policy files of 0, 5, 10, 25, 50, 100, and 200 rules such that in all files, only the last rule would ever match. Therefore we are actually measuring the effect of checking the rules, not simply loading a larger ruleset, which would have almost no effect. For each policy file, we then used SoapUI to send 8 Kb messages to xESB using 100 parallel threads for 3 minutes. We repeated this process 3–5 times. Since SoapUI does not return the RTT for individual messages, we looked at the average RTT. The results are plotted in Fig. 9, left. The most important conclusion is that xESB is almost unaffected by the size of the policy file: the main delay seems to be in message processing, not policy enforcement. In fact, profiling shows that policy enforcement takes only about 0.2% of CPU time.

The next question we asked is "How does the number of parallel clients affect the RTT?". Using a similar method as above (8 Kb messages using 100 rules for 3 minutes, repeat 3–5 times, then look at the average RTTs), we arrived at Fig. 9, right. As expected, both the average RTT and the variability rise linearly.

A curious feature that can be seen in the figure is the tremendous variability in RTT. Since this variability also shows in the uninstrumented ServiceMix, we conclude that policy enforcement is not responsible for this. We conjecture that this is due to the staged-event architecture of ServiceMix, where processing is done in bursts because computation happens in stages[4].

[4] Staged Event-Driven Architecture `http://www.eecs.harvard.edu/~mdw/proj/seda/`

9 Related Work

There is a large number of security standards that cover XML message validation, authorization, encryption or even federated authentication: OASIS's WS-Security, SAML, XACML[5], and other WS-* specifications. These standards only deal with particular narrow issues of SOA scenarios (point-to-point authentication, authorization, message integrity, etc.) and not with SOA policy compliance.

The work on enforcing security at the message-level is limited to considering access control policies. The solution of Svirskas et al. [7] is limited to controlling service access and logging on the ESB. A similar approach [8] suggests several infrastructure security services to act on different types of events by means of a gateway, but the separation between business and security concerns is not clear. Maierhofer et al. [9] describe a dynamic enforcement framework for security at the message-level, but do not discuss interoperability nor implementation. Another solution [10] suggests a custom security service bus for enforcement of complex policies. Other conceptual approaches based on law-governed interactions [11] aim to model enforcement of laws onto communication between servers and clients, but they consider generic distributed dedicated entities to perform the law realization. Our assumption differs in that it uses the ESB as a centralized mechanism that either performs on-the-fly enforcement or delegates it to a trusted entity. We explicitly focus on access and usage control policies, and unlike other approaches, we offer a concrete model and a proof-of-concept implementation.

Concerning the policy language, elaborate access control languages (e.g., Ponder [12], EPAL [13], SPL [14]) are unable to express obligations that pertain to usage control. More generic usage control centric languages are POLPA [15] and OSL [16]; while the former does not explicitly address obligations, the latter is not supported by an implementation. Unaware of any implementation of a generic usage control language that supports compensations, we have developed a proof-of-concept policy language fit for the ESB. Compared to theoretic works on access control compensations [17], violation management [18] and obligation assessment [19, 20], we go beyond access control and provide an implementation of compensations on the fly. This means that whenever a violation of some service usage rule is detected, the correction happens as the event travels through the system, before it reaches some interested party.

In usage control enforcement [21], Katt et al. [22] add the notion of post-obligations to the obligation model; they consider and implement a mechanism for ongoing enforcement. The work of Pretschner et al. [5, 23] describes a formalized usage control language and the mechanisms to enforce such a language, but do not cover an enforcement model for SOA. xESB reuses these enforcement mechanisms as well as idea of post-obligations but applies them for the first time to the ESB level.

[5] These OASIS standards can be found at `http://docs.oasis-open.org`

10 Conclusion and Future Work

The paper presented xESB, an instrumented JBI ESB for the enforcement of security policies that are organization-wide. xESB is able to enforce both access and usage controls policies. The rich enforcement semantics of xESB allows not only to reject ESB messages that violate a policy but also to compensate that violation. xESB also introduces and supports indicators aiming to help the security administrator analyze and derive useful information about policy violations (e.g., discover configuration mistakes) and their impact to the overall security of the organization. While initial performance tests are very promising, we are planning to use xESB with a commercial SOA application and run more extensive tests to validate the initial results.

We are currently working to extend xESB on four aspects:

Optimization. In a large policy file a lot of time is spent evaluating conditions in order to find those rules or obligations that apply to a given message. To improve this, we will implement some form of the Rete algorithm [24], or select rules according to which message parts appear in `where`-clauses.

Conditions on message payload. We will extend xESB so that complex conditions on the message payload can also be evaluated. We will most likely base this capability on XPath.

Performance measurements. Apart from measuring the performance impact of the executor and modifying mechanism, we plan to evaluate different policy engines against our enforcement design, thus checking its extensibility.

Performance indicators. We will implement performance indicator support in order to derive general information on the runtime enforcement process (e.g., statistics on the rules that are frequently enforced or violated).

References

1. Zimmermann, O., Doubrovski, V., Grundler, J., Hogg, K.: Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2005) 301–312
2. UK Government: The privacy and electronic communications (ec directive) regulations 2003. `http://www.opsi.gov.uk/si/si2003/20032426.htm` (June 2009)
3. European Parliament: Directive 95/46/ec of the european parliament and of the council. `http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf` (June 2009)
4. Sun, Java Community Process Program: Sun JSR-000208 Java Business Integration. `http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html` (August 2005)
5. Pretschner, A., Hilty, M., Basin, D., Schaefer, C., Walter, T.: Mechanisms for usage control. In: Proc. ASIACCS '08, New York, NY, USA, ACM (2008) 240–244
6. Ueno, K., Tatsubori, M.: Early capacity testing of an enterprise service bus. In: ICWS '06: Proceedings of the IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2006) 709–716

7. Svirskas, A., Isachenkova, J., Molva, R.: Towards secure and trusted collaboration environment for european public sector. Collaborative Computing: Networking, Applications and Worksharing, 2007. CollaborateCom 2007. International Conference on (Nov. 2007) 49–56
8. Leune, K., van den Heuvel, W.J., Papazoglou, M.: Exploring a multi-faceted framework for soc: how to develop secure web-service interactions? Research Issues on Data Engineering, Proc. 14th Intl. Workshop on (March 2004) 56–61
9. Maierhofer, A., Dimitrakos, T., Titkov, L., Brossard, D.: Extendable and adaptive message-level security enforcement framework. Networking and Services, 2006. ICNS '06 (2006) 72–72
10. Goovaerts, T., De Win, B., Joosen, W.: Infrastructural support for enforcing and managing distributed application-level policies. Electron. Notes Theor. Comput. Sci. **197**(1) (2008) 31–43
11. Lam, T., Minsky, N.: A collaborative framework for enforcing server commitments, and for regulating server interactive behavior in soa-based systems. In: Proceedings of the5th International Conference on Collaborative Computing: Networking, Applications and Worksharing. (November 2009) 1–10
12. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks, Springer-Verlag (2001) 18–38
13. Backes, M., Pfitzmann, B., Schunter, M.: A toolkit for managing enterprise privacy policies. In: In Proc. of ESORICS03, LNCS 2808, Springer (2003) 162–180
14. Ribeiro, C., Zqute, A., Ferreira, P., Guedes, P.: Spl: An access control language for security policies with complex constraints. In: In Proceedings of the Network and Distributed System Security Symposium. (1999) 89–107
15. Baiardi, F., Martinelli, F., Mori, P., Vaccarelli, A.: Improving grid services security with fine grained policies. In: Proc. On the Move to Meaningful Internet Systems Workshop. Volume 3292 of LNCS., Springer-Verlag (2004) 123–134
16. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In: 12th European Symposium on Research in Computer Security (ESORICS 2007). Volume 4734 of LNCS., Springer-Verlag (2007) 531–546
17. Povey, D.: Optimistic security: A new access control paradigm. In: In Proceedings of 1999 New Security Paradigms Workshop, ACM Press (1999) 40–45
18. Brunel, J., Cuppens, F., Cuppens, N., Sans, T., Bodeveix, J.P.: Security policy compliance with violation management. In: FMSE '07, New York, NY, USA, ACM (2007) 31–40
19. Irwin, K., Yu, T., Winsborough, W.H.: Assigning responsibility for failed obligations. IFIP Intl. Federation for Information Processing **263** (2008) 327–342
20. Irwin, K., Yu, T., Winsborough, W.H.: On the modeling and analysis of obligations. In: CCS '06, New York, NY, USA, ACM (2006) 134–143
21. Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. **7**(1) (2004) 128–174
22. Katt, B., Zhang, X., Breu, R., Hafner, M., Seifert, J.P.: A general obligation model and continuity: enhanced policy enforcement engine for usage control. In: Proc. SACMAT '08, New York, NY, USA, ACM (2008) 123–132
23. Pretschner, A., Schütz, F., Schaefer, C., Walter, T.: Policy evolution in distributed usage control. In: 4th Intl. Workshop on Security and Trust Management. (06 2008)
24. Forgy, C.: A network match routine for production systems. Working paper, Carnegie-Mellon University (1974)