

User Data Confidentiality in an Orchestration of Web Services

Thomas Demongeot¹, Eric Total², Valerie Viet Triem Tong² and Yves Le Traon³

¹DGA - Information Superiority Unit - Bruz - France
Telecom Bretagne - Cesson-Sévigné - France
thomas.demongeot@dga.defense.gouv.fr

²Supelec - Cesson-Sévigné - France
surname.name@supélec.fr

³University of Luxembourg
yves.letraon@uni.lu

Abstract:

Web Services are currently the base of a lot of e-commerce applications. Nevertheless, the clients often use these services without knowing anything about their internals. Moreover, they have no clue about the use of their personal data inside the global applications. BPEL (Business Process Execution Language) is a programming language for orchestrating Web Services within Service-Oriented Architecture (SOA). As one feature of SOAs is the dynamic discovery of services actually used during execution, a BPEL user does not know prior to the execution how, and by who, the data he provides will be used. In this paper, we offer the opportunity to the user to specify constraints on the use of its personal data. To ensure the privacy of data at runtime, we define a distributed security policy model. This policy is configured at runtime by the user of the BPEL program. This policy is enforced within a BPEL interpreter, and ensures that no information flow can be produced from the user data to unauthorized services. However, the dynamic aspects of the web services lead to situations where the policy prohibits the nominal operation of the orchestration (e.g., when using a service that is unknown by the user). To solve this problem, we propose to the user to dynamically permit exceptional unauthorized flows. In order to make its decision, the user is provided with all information necessary for decision-making. An implementation inside the Orchestra BPEL interpreter illustrates our approach and exhibits the CPU overhead induced by the security mechanisms.

I. Introduction

Web services [1] were originally designed as a set of reusable services freely available to everyone. Service-orientation eventually offers an elegant way to build new services composed of existing ones using the notion of orchestration.

On one hand, since services are based on encapsulation, the client does not need to understand how a service works. On the other hand, this lack of information also means that the client does not know how his data are used and by who. Currently, most of the efforts in web service security focus on the confidentiality of the information at the communica-

tion protocol level, but do not solve the problem of how to make a specific service orchestration trustable for the clients. Even if the service orchestration provider is trustable, it has no technical solution to guarantee for a specific client that it satisfies his expectations in terms of data protection. User data protection in a service orchestration is thus crucial, and requires two basic bricks. First, the expectations of the client must be expressed, which implies some security policy language to be available. In this paper, we propose such an elementary data protection policy configurable by the user of the service.

Second, the technical support for checking the client's data protection policy must be embedded in the orchestration interpreter. In this paper, we propose to check at runtime whether the data protection policy is satisfied with a prototype tool called OrchestraFlow. This tool extends a BPEL interpretation engine, which is the standard language for programming an orchestration of web services. BPEL (Business Process Execution Language) [2] is a relatively simple language that describes the sequences of service calls necessary to achieve properly a composite service. A BPEL program¹ may receive information from users, and use these data to provide information to the invoked services. Therefore, the BPEL program produces information flows from the user data to the used services. The problem is whether these information flows are legal according to the user privacy policy. OrchestraFlow takes the data protection policies of the service actors as inputs and checks whether there is a risk of information leakage at runtime w.r.t. the policy. Instead of a static analysis of the BPEL program, a dynamic analysis has been chosen in order to be able to handle dynamic function discovery and dynamic update of the security policy.

Before describing this solution, we first propose an example to illustrate the problems of confidentiality that may occur in a BPEL program (Section II). The solution proposed

¹We define a BPEL program as a Web Service written in BPEL executed by a BPEL interpreter. A BPEL program is an orchestration of Web Services.

here is based on dynamic information flow tracking, thus Section III presents a state of the art on this topic. We define a security policy that specifies legal information flows. We define what properties it can provide (Section IV) and how to verify the policy (Section V). Section VI presents how to dynamically update the security policy. Finally we describe our implementation called OrchestraFlow (Section VII) and study its performances (Section VIII). Finally, we conclude and expose future work in Section IX.

II. Problem statement

A web service orchestration consists in the execution of a set of services that manipulate and transform data. These data are injected by other services or by the users. In a BPEL program these data are protected at different levels. At message level WS-Security [3] aims at providing security for exchanging SOAP messages. Besides security architecture, there exists XML-based languages such as SAML (Security Assertion Markup Language) [4] and XACML (eXtensible Access Control Markup Language) [5] that allow to specify access control rules for accessing data or services. However there is no access control to data once the data has left its original container. Using XACML we can specify that a user or a service can access some data but once these data are accessed by a service there is no control on their propagation. In this article we aim at providing a better security level by offering both a context-adaptative security policy driven by users and a dynamic enforcement mechanism of the security policy.

Before detailing our approach, let us have an example: Figure 1 details an online bookshop service which uses different services, such as a bookstore and a bank providing a payment service. The accesses to these services are orchestrated in a BPEL program.

The various actors in this transaction are:

- the seller who provides three services: it computes the total amount of the transaction to allow bank payment (s_1), emits the bill (s_2) and finally prepares the order that will be delivered to the client address (s_3);
- the bank provides a payment service (s_4);
- the client who could be seen as a service (s_5).

The services are not necessarily known before the execution as they can be discovered at runtime, after a search in a directory of services for example. Thus we don't know before the execution which services are called. Due to this particular feature we believe that a precise security mechanism depends on the context of the execution and has to be adjustable at run-time.

As we will explain in the later our approach will permit the client to specify that only the seller knows which product the client has chosen, the bank account information is provided only to the bank. The security policy will be defined by users and can be updated at run-time, for instance when the services are discovered. The security policy enforced at runtime relies on information flow tracking mechanisms that permit to detect user data leakage inside a BPEL interpreter.

In the following section we briefly present work related to information flow tracking in programs.

III. Related work

The area of information flow tracking has been well-studied during the last decade. The basic idea of information flow tracking is that sensible data are marked with an identifier sometimes called a taint, a label, a tag or a mark. The marks are propagated along the flow to taint objects in the system. The propagation can be either dynamically observed or statically analysed. Several researches have helped to strengthen the control of data privacy in BPEL programs, particularly by statically controlling data flows. In [6] BPEL is considered as the description of a distributed collaborative system with a multi-level security policy. This policy ensures that data from Web Services are used properly, it lacks flexibility and does not manage dynamic adaptation. [7] and [8] proposed type systems in order to guarantee non-interference property in dynamic service composition. But the method proposed by [7] needs to analyse each service involved in the orchestration and does not support complex orchestration. In [8] each service involved in the orchestration need to produce a contract describing its internal behaviour and the authors proposed a framework to analyse service orchestration. In [9], the authors propose an XML schema for specifying an employment policy of available Web-Services statically verified in BPEL programs. In both cases, security policies are defined by the host of BPEL and do not specify a security policy for each user. Moreover, the verification of information flows is done statically: it is impossible to address the problem of dynamic discovery of services. In [10] and [11] Myers and Liskov propose more expressive marks (which are called labels). A label attached to a value denote both owners and readers of this value. An owner decides which principals can access his data, these principals are *the readers*. In [12] Myers presents Jif, where labels are used to annotate data items in a Java program. Jif checks at compile time, in a manner similar to type checking, if all the executions of annotated programs verify the information flow policy. In their approach, the information flow policy consists of the definition of the readers by the owner. This policy is defined before the analysis and can be updated by relabelling data. Their model authorises only two relabelling rules: restriction and declassification. Data can only be relabelled from L_1 to L_2 if L_2 is more restrictive than L_1 intuitively if it removes readers, adds owners, or both. A datum is declassified when it is relabelled to a label containing more readers for an owner o or when a owner o is removed. A declassification process is allowed only when the process acts for o . In [13, 14] the authors explicitly distinguish information from containers and thus propose to mark containers of information with two tags reflecting both the origins of the value and the security policy attached to the container. More precisely sensible data are associated to a numerical identifier and an information flow policy specifies how combinations of these identified data can flow in information containers. The model of marks presented in [14] can be either implemented at system level or at program level. In [13] information flows are tracked at run-time allowing us to check if the current execution is correct with regard to the definition of the policy. The policy is completely defined at the initialisation and can be either deduced from an interpretation of access control rights or manually defined. The policy can be updated at run-time

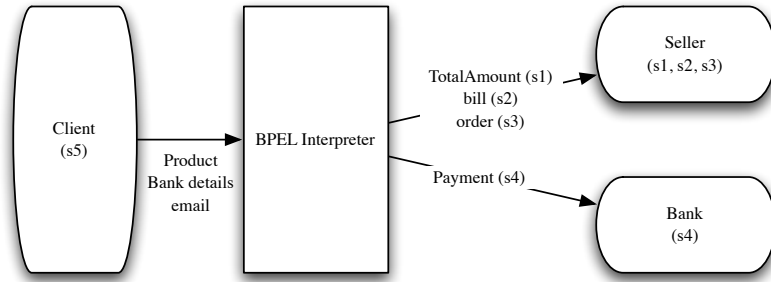


Figure 1: Online bookshop service

simply by changing the tags. In [14] the authors explain how to perform a modification of the policy by changing tag value but do not define how, why or when to perform such a modification. We propose to adapt these previous models in the particular context of web services. We aim to observe information flows inside an orchestration of web services in order to ensure the user's data protection. We adopt a dynamic observation of these flows since in a context of web services we will dynamically discover the environment. As in [13, 14] we explicitly identify user's data with numerical identifier. As Myers and Liskov in [10] and [11] the security policy will specify owners and readers of the identified information items. In other words a user defines which services can access his information items. The description of all readers could be difficult for uninformed users. To solve this problem we propose to dynamically update the security policy when services are discovered. Our tool interacts with the users to adapt or complete the security policy when required.

IV. Privacy Security Policy

A. Characterization of information and information granularity

A piece of information is a data item, a value such as a string, or an integer. A piece of information is provided to a web service orchestration through a call to this service. This piece of information is manipulated by the orchestration and the services it invoked and mixed with other pieces of information. In this work, we consider that sensitive information and in particular users private data have to be monitored in order to protect where these information data items flow. For that purpose we reuse the notion of *atomic information* first introduced in [14] to identify sensitive or private information. Any piece of information handled in the system is either atomic or obtained after treatments (like calculus) on one or more atomic information. Here any non-atomic information is the compound of one or more atomic information. For example, if x, y are atomic information $2 \times x, x + y, \dots$ are compound information, the first non-atomic information results from the use x , the second results from the use of x and y .

In the example detailed in section II, atomic information items are provided by the client : the chosen product, bank details and client email address. These atomic data are used to compute all information items handled by the complete

system, such as the total amount of the transaction, the confirmation of payment, final product delivery notification,...

In a web service orchestration, the information is located in logical containers of information like the variables manipulated by services. The operations performed by programs or services will generate information flows between variables and consequently information will be mixed and/or will move from one variable to another. In this work we want to prevent private or sensitive information to be accessed by a non-authorized service, i.e., we want to ensure that sensible information flows only into variables readable by authorized services.

B. Defining the security policy

The security policy allows the user to specify which services are authorized to manipulate each atomic information (and by composition for all the compound information). For that purpose we first determine an *owner* for each atomic information (usually the service/user that provides it to the system). The owner is responsible for determining statically (at the start of the service invocation) or dynamically (during the execution of the service orchestration) the set of services that can access this information. These services will be called *information readers*. A service is allowed to read an atomic information only when it appears in the set of legal readers for this atomic information.

The rest of the policy is determined by composition. When an information is derived from several atomic information items, the owner of this compound information is the set of all owners of atomic information. The readers of this compound information are all the services that are also readers of each atomic information from which it derives.

This security policy can be seen as an information flow policy: a flow of information i (atomic or compound) to a container belonging to a service s is legal if and only if the service s has the right of access to information i , i.e., if s is a reader of i .

More formally we use the following notations:

- **Information:** $I = \{i_1, \dots, i_n\}$ is the set of atomic information of the system. Information derived from several atomic information i_j, \dots, i_k is denoted by $i_j \oplus \dots \oplus i_k$
- **Services:** $S = \{s_1, \dots, s_m\}$ is the set of services of the system.

- **Owners** of information i are services that we denote $owner(i) \subseteq S$. They are defined as follows:

- If i is an atomic information then its owner is the service that injected it into the system.
- If i is a compound information, i.e., $i = i_j \oplus \dots \oplus i_k$ then

$$owner(i) = owner(i_j) \cup \dots \cup owner(i_k) \quad (1)$$

- **Readers** of an information i are services defined by the owners of i which we denote $readers(i) \subseteq S$. Readers are defined as follows:

- if i is an atomic information, readers of i are the readers allowed by the service which injected it into the system;
- if $i = i_j \oplus \dots \oplus i_k$ then

$$readers(i) = readers(i_j) \cap \dots \cap readers(i_k) \quad (2)$$

- **The security policy** defines allowed readers for each atomic information, rules of composition (1) and (2) define, by composition, readers of every compound information. The policy is defined by the owners of information, since an owner determines the readers that are allowed to read its atomic informations. A call to a service that brings an information is legal only if the service called is a *reader* for this information. In the same way, a response from a service is only authorized if the caller is a legal reader for the information received.

The policy can be updated at any time by adding or removing a reader from the set of readers of an information. An owner is responsible for removing readers to its own atomic information. When an information is compound, the several owners have to agree for any modification.

C. Example of privacy policy

Let us consider again the example detailed in Section II in Figure 1. In this example five services are present:

- s_1, s_2 and s_3 are three services provided by the seller;
- s_4 is the payment service provided by the bank;
- s_5 is the user service that calls the BPEL orchestration to place an order.

Atomic information items in the system are provided by the user service, i.e., service s_5 . i_1 corresponds to the chosen product. The client imposes that i_1 is accessible only to the seller and thus $readers(i_1) = \{s_1, s_2, s_3\}$, the services provided by the seller. i_2 corresponds to the bank details. The client imposes that they are accessible only to the bank, we thus have $readers(i_2) = \{s_4\}$, the payment service provided by the bank. Similarly the client wants i_3 (the client email address) to be accessible to the seller only. We have $readers(i_3) = \{s_1, s_2, s_3\}$. In all cases the owner of such atomic information is the service calling the command (called directly by the client), i.e., s_5 .

More formally in this example

Atomic information name	Owners	Readers
i_1	s_5	$\{s_1, s_2, s_3\}$
i_2	s_5	$\{s_4\}$
i_3	s_5	$\{s_1, s_2, s_3\}$

Figure 2: A security policy for the example detailed in section II

- **Informations** are $I = \{i_1, i_2, i_3\}$;
- **Services** are $S = \{s_1, \dots, s_5\}$;
- **Owners** of I , $owner(I) = owner(i_1) = owner(i_2) = owner(i_3) = \{s_5\}$;
- **Readers** of I are defined as follows, $readers(i_1) = \{s_1, s_2, s_3\}$, $readers(i_2) = \{s_4\}$, $readers(i_3) = \{s_1, s_2, s_3\}$.

In other words, the policy is entirely defined Figure 2.

V. Dynamic Checking of the Security Policy

In this work, the security policy is enforced through meta-data or simply labels put on every containers of information: which means on every variables in a BPEL program. As it has been proposed by Myers in [10] a label of a variable denotes the owners and the legal readers of its content. In order to follow the origin of information flow, we add to each variable the list of initial information used to produce the content of this variable. The value of a label is initialized as empty and is firstly modified when a new item of information is injected in the web service through a call to this service. At this moment, the injected item of information is considered as atomic, its owner is the caller. The caller also defines the allowed readers for this new item and consequently the new value of the label. The label is further modified at each operation on the variable that modify the content of the variable. Labels are modified to reflect owners and readers attached to the information contained in the variable. When a service calls another service or makes a response to another service, a verifier checks if the flow engendered is legal with respect to the current security policy. More precisely the verifier checks if the recipient of the flow appears as a reader in the label of the item sent. In the following, we formally define how labels are defined and modified.

A. Definition of Security Labels

As stated before, a label is a meta-data attached to each container and describes owners and readers of the information currently located in the container. If c is a container its security label is of the form

$$L_c = \{\mathbf{i}_1 : \mathbf{s}_\alpha \triangleright s_{\alpha_1}, \dots, s_{\alpha_n}; \dots; \mathbf{i}_j : \mathbf{s}_\beta \triangleright s_{\beta_1}, \dots, s_{\beta_m}\}$$

Such a label means that information i contained in c is based on information $\mathbf{i}_1, \dots, \mathbf{i}_j$. Information \mathbf{i}_1 is owned by $owners(\mathbf{i}_1) = \mathbf{s}_\alpha$ which authorizes readers $s_{\alpha_1}, \dots, s_{\alpha_n}$. Depending on this label the readers allowed to access the information located in c are those authorized by all the owners,

i.e., $readers(c) = \{\{s_{\alpha_1}, \dots, s_{\alpha_n}\} \cap \dots \cap \{s_{\beta_1}, \dots, s_{\beta_m}\}\}$.
By abusing the notation we may use $owners(L_c)$ or $readers(L_c)$ to express the owners/readers of a container c labeled by L_c .

B. Initialization and Modification of Security Labels

Let us consider a service s_1 injecting an item of information i in another service s_2 by calling s_2 using a variable v . The service s_1 is considered to be the owner of the atomic information i now located in the variable v of s_2 . The variable v is the container of i and its label is on the form $\{s_1 \triangleright s_{\alpha_1}, \dots, s_{\alpha_n}\}$ where $s_{\alpha_1}, \dots, s_{\alpha_n}$ are the readers of i allowed by s_1 . In practical terms if the service s_1 is executed by a user, this user will be asked to define the services allowed as readers of its own information.

When a service is called, it makes some internal computation before sending a response. These internal computations induce some information flows and modify the content of containers of information. Since a label attached to a container describes the security policy of its current content, it has to be updated at each observation of an information flow towards the container.

From a general point of view, we consider a set of containers c_j, \dots, c_k labeled by L_j, \dots, L_k if we observe an information flow from the containers c_j, \dots, c_k to another container c , then we update the label of c which is now the union of labels attached to c_j, \dots, c_k . As Myers, we use the notation $L_j \sqcup \dots \sqcup L_k$ to denote the union of labels. The precise definition of \sqcup is given below. This new label means that the owner of the content of c is now the union of owners of content located in c_j, \dots, c_k and the readers are those commonly allowed by these owners. The new label should also reflect that information contained in c depends on information from c_j, \dots, c_k , i.e., the label should reflect the information history.

Labels for Derived Values (Definition of $L_1 \sqcup L_2$)

$$\begin{aligned} owners(L_1 \sqcup L_2) &= owners(L_1) \cup owners(L_2) \\ readers(L_1 \sqcup L_2) &= readers(L_1) \cap readers(L_2) \\ history(L_1 \sqcup L_2) &= history(L_1) \cup history(L_2) \end{aligned}$$

Let us have an example with three containers c_1, c_2 and c_3 , respectively labeled by :

- $L_{c_1} : \{\mathbf{i}_1 : s_1 \triangleright s_5, s_6; \mathbf{i}_2 : s_1 \triangleright s_5, s_6\}$
- $L_{c_2} : \{\mathbf{i}_1 : s_1 \triangleright s_5, s_6; \mathbf{i}_3 : s_2 \triangleright s_6, s_7\}$
- $L_{c_3} : \{\mathbf{i}_4 : s_3 \triangleright s_4, s_7\}$

We consider an information flow from c_1 and c_2 to c_3 . This flow modifies the content of c_3 which is now a value derived from those located in c_1 and c_2 . The label L_{c_3} is updated to $L_{c_1} \sqcup L_{c_2}$, i.e.

$$\{\mathbf{i}_1 : s_1 \triangleright s_5, s_6; \mathbf{i}_2 : s_1 \triangleright s_5, s_6; \mathbf{i}_3 : s_2 \triangleright s_6, s_7\}$$

and means that

$$\begin{aligned} owners(c_3) &= owners(c_1) \cup owners(c_2) = \\ & \{s_1\} \cup \{s_1, s_2\} = \{s_1, s_2\} \\ readers(c_3) &= readers(c_1) \cap readers(c_2) = \\ & \{s_5, s_6\} \cap \{s_5, s_6\} \cap \{s_5, s_6\} \cap \{s_6, s_7\} = \{s_6\} \\ history(c_3) &= history(c_1) \cup history(c_2) = \\ & \{i_1, i_2\} \cup \{i_1, i_3\} = \{i_1, i_2, i_3\} \end{aligned}$$

The definition of the security policy is carried out via the propagation of the labels attached to the containers of information. When a service performs a response using a variable c this response will be authorized according to the security policy if the recipient appears as a reader in L_c .

From a practical point of view, in our work the security policy is propagated through the labels at runtime in a modified BPEL interpreter. The legality of a call to a service or a response from a service is checked just before the call / response.

VI. Dynamic Update of the Security Policy

Let us consider a BPEL program performs a call of a service s (or similarly a response to a service s) using data d having a label on the form

$$L_d = \{\mathbf{i}_1 : s_1 \triangleright s_{1_1}, \dots, s_{1_n}; \dots; \mathbf{i}_j : s_j \triangleright s_{j_1}, \dots, s_{j_m}\}$$

We have to verify if this call is legal with regard to the security policy before performing the call. By definition of the security policy this call is legal if and only if the service s is an authorised reader for the data d . To check this legality we only need to verify if s appears as a reader in the label attached to d , i.e., if $s \in \{s_{1_1}, \dots, s_{1_n}, \dots, s_{j_1}, \dots, s_{j_m}\}$. If s is an authorized reader then the BPEL program performs the call. Otherwise we ask owners of s to confirm if that the call must although be authorized. Indeed, since services can be dynamically discovered we can not decide if the call is really forbidden or if the owners have not completely defined the security policy.

We use a dedicated service to ask all owners (s_1, \dots, s_j) if they authorize or not to send a compound information d computed using their atomic information resp. $\mathbf{i}_1, \dots, \mathbf{i}_j$.

More precisely the BPEL interpreter calls a dedicated service to contact owners of information. This service is an exception to the security policy, we consider that this particular service is a reader for any atomic information. In future work we plan to protect this dedicated service: for instance we plan to encrypt the data send to/by this service. This service is used to ask every owner s_k of atomic information \mathbf{i}_k if they accept to modify the policy of \mathbf{i}_k . The service thus uses a request composed of four parts:

- the initial information i_k that was used to compute the value d ;
- the value d if the owner is an authorized reader of d , this part is empty otherwise;
- the service s ;
- if the information actually sent to the service depends explicitly or implicitly on the initial information.

For each owner, this call may have in three possible responses:

- **(refusal)** the owner refuses to modify the security policy.
- **(temporary exception)** the owner accepts the update of the security policy only for this call/response of service.

- (**agreement**) the owner accepts the update of the security policy until the end of the execution of the BPEL program. In this case the label of the variable is modified.

If at least one owner refuses the modification, the service call (or the response) is not performed. If all the owners accept the modification but at least one of them authorises only a temporary exception then the call (or the response) is performed and the label attached to d remains the same. Finally when all the owner accept the modification, the label is modified: s is added as reader for d .

VII. OrchestraFlow : an implementation of a privacy policy in a BPEL interpreter

In this section we present OrchestraFlow which implements the model detailed in the previous sections as a patch for the BPEL interpreter Orchestra². OrchestraFlow taints variables of a BPEL program using labels as detailed before, the implementation of labels is presented in section VII-A. A label is updated at each modification of the content of the variable. In a BPEL program this content is directly modified by operations involving the variable. Thus we have modified the original Orchestra interpreter to observe information flows made by a BPEL program and to consequently update the labels of the involved variables. This part of the implementation is detailed in section VII-B. In Section VII-C and VII-D we present mechanisms that allow dynamic checks and updates of the security policy.

A. Security Label Implementation

A BPEL program takes as inputs messages coming from other web services. Because all messages are in XML format, we modify the XML inputs in order to add our security label. We modified all XML primitive types by adding an optional label attribute where authorized readers are represented by an URI (adress of Web Services) separated by a semi-colon. If the label attribute is used with an empty string then no service is allowed to access that data. If the label attribute is not used, all services are allowed to access that data. In the following item the chosen product is accessible by the online-shop and the bank, bank details are accessible by nobody, and the e-mail adress by everybody.

```
<chosenProduct
  readers="http://myShop ; http://myBank">
  myProduct
</chosenProduct>
<bankDetails readers=""> 123456789 </bankDetails>
<adress> myAdress </adress>
```

In order to allow dynamic update of the security policy, each user of a BPEL program uses a client side security service. The security service is a simple web-service that runs on the computer of the client. This service receives all requests to update the security policy defined in the BPEL program.

If the sender is another web service which does not execute OrchestraFlow, then we consider the variable as a new atomic information without label (meaning that all services are legal readers). Applying this property allows us to be

Example of a BPEL variable in a XML tree structure:

```
<Payment>
  <amount>12</amount>
  <bankDetails >123123</bankDetails>
</Payment>
```

and its corresponding label added in OrchestraFlow

```
|--userData
  |--amount : Label :
    {[Product : http://localhost/Bookstore :
      http://localhost:8081/Seller/ ;
      http://localhost:8081/Bank/]}
  |--bankDetails : Label :
    {[bankDetails : http://localhost/Bookstore :
      http://localhost:8081/Bank/]}
```

Figure. 3: Example of a BPEL variable and its label

compatible with existing BPEL interpreters that do not carry out our protection mechanisms.

In OrchestraFlow, a label is thus a list of triple on the form (initial information ; owner ; list of readers authorized for this owner). In BPEL, a variable is represented via a XML tree structure that can be composed of leafs (simple elementary values) or nodes (complex variables composed of several elementary values). In order to store the labels attached to each variable, the tree structure is duplicated and filled up with the labels of the elements composing the variable.

Figure 3 is an example of a variable composed of two parts (amount and bankDetails), each of them has its own label stored on a duplicated tree structure. After the initial information, the first URI of a label represents the owner of the data, and the following URIs, separated by a semi-colon, represent the authorized readers of this data. Amount is produced by the initial information product. It has one owner which authorizes two readers to access this data. BankDetails has the same owner which authorizes one reader.

B. Propagation of Labels

The label of a variable is updated at each observation of an information flow. As defined early by D. Denning in [15], *Information flows from object x to object y , whenever information stored in x is transferred to, or used to derive information transferred to, object y .* We distinguish here *implicit or explicit information flow*. An *implicit information flow signals information through the control structure of a program* [16]. Our reader will find a complete survey on this subject in [16]. First we focus on explicit information flows between variables which are transfers of information induced by operations made by the program involving these variables. In BPEL the operations are listed on table 1. Among them Assign, Invoke, Receive, Reply induce explicit information flows. Later we also treat implicit flows induced by conditional and loop.

OrchestraFlow extends Orchestra in order to update concerned labels at each call of one of the mentioned operations.

1) Production of explicit information flow

Explicit information flows are mostly induced by assignments and communication with services.

An assignment copies the value of the expression e in x . After the execution of the assignment the information contained in x depends now on every information contained in

²<http://orchestra.ow2.org/xwiki/bin/view/Main/WebHome>

	Order	Description	Security Policy		Information Flow	
			Input	Verification	Explicit	Implicit
Assignments	Assign	Assignment			X	
Empty activity	Empty	Empty activity				
Communication with services	Invoke	Web Service call	X	X	X	
	Receive	Reception of an incoming message	X			
	Reply	Response to an incoming message		X		
Sequence	Sequence	Sequential organization of activities				
Conditional and loops	Switch	Conditional execution of activities				X
	While	Loop execution of a set of activities				X
	ForEach	Loop can run in parallel mode				X
Exceptions	Compensate	Compensation				
	Terminate	Explicit request to stop a process				X
	Throw	Raising an exception				X

Table 1: The main orders of BPEL

e. In this case, we must ensure that the label of x after the execution of the assignment reflects the policy of the information contained in e . When e is simply a single BPEL variable then value of label of x is updated to the value of the label of e . In other words, if an assignment copies the value of e in x then OrchestraFlow propagates the value of the label \mathcal{L}_e in \mathcal{L}_x . More generally an expression e in a BPEL program could be a part of a BPEL variable or a more complex expression written in an external language. OrchestraFlow uses, like Orchestra, XPath 1.0 as expression language. For each XPath expression we calculate the resulting label from each information contained in the XPath expression according to the definition V-B.

Communication between Services Three BPEL functions allow communications with external services: `invoke`, `receive` and `reply`. The first, `invoke`, provides synchronous communications with services, i.e., in the same function data are sent to the service and a response is received. In order to allow asynchronous communication with services, we use the same function `invoke` with the second function `receive` which allows the asynchronous reception of the response of the service called with the `invoke` function.

These communication primitives produce information flows from the caller to the receiver. It is thus necessary to update the labels of the messages sent (case of `invoke`) or the labels of the variables assigned at the reception of a message (case of a `receive`) by performing the union of the labels of the data involved.

For example, by using an `invoke` function, the service `my_service` is called with the variable e as input parameter. The result of this service will be stored in the variable x . The variable x after executing the service depends both on the information returned by the called service (`my_service`) but also on information contained in the variable e . Indeed there are information flows from e and the return of `my_service`. The security label of x after the execution of the invocation of `my_service` is computed according to the definition V-B.

In the same way we propagate labels in OrchestraFlow during an asynchronous service call with the functions `invoke` and `receive`.

2) Production of implicit information flow

The second type of information flow that can be created by the language is of implicit type. It is what happens by example during conditional operations and loops. In these cases, data manipulated within the structure of the loop or conditional depends on the variables used in the conditional statement of the condition or the loop.

Loops and conditionals are treated in the same way. All operations performed inside the conditional or the loop are implicitly dependent on the value of the condition c .

In the case of assignments in a conditional, the value of the variable x receiving the expression e also depends on the value of c . There is an information flow from c to x . The label of x is computed from labels of e and c according to the definition V-B.

In the case of service invocations in a conditional, if a service call is performed there is an implicit information flow from c to the service call since it is done according to the value of c . We must, at the time of the service call, ensure that it is also authorized by the security policy associated with c .

In OrchestraFlow we modified the `ScopeRuntime` class in order to add a stack which contains labels of conditional or while condition. When a conditional starts, a label is added to this stack. At the end of this conditional the label is removed.

During the execution of an explicit flow the computation of the new label takes care of both the labels of the expression considered in the explicit flow and the resulting label of the implicit flow stack.

C. Checking the Security Policy

The legality of the information flow is checked when a service tries to send information to an other service. Two functions send data to external services: `invoke` and `reply`. When one of this function is called, we verify that the service call complies with the security policy, i.e., if the recipient service belongs to the authorized readers of the data. More formally, when a service uses `invoke` or `reply` with output variable m towards a service s OrchestraFlow checks if $s \in \text{reader}(m)$ as defined in definition V-B.

In order to prevent implicit information flow, a second verification must be done. The service call should be authorized by the resulting label of the implicit flow stack.

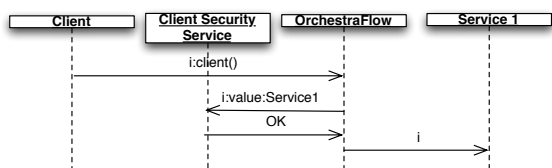


Figure 4: BPEL program with security service

	Example 1 Simple label transmission	Example 2 While loop	Example 3 Service call
Orchestra	29,33 ms	35,95ms	45,63 ms
OrchestraFlow	34,12 ms	45,62 ms	62,12 ms
Overhead	16%	27%	36%

Table 2: CPU overhead

D. Dynamic update of the security policy

When an illegal flow is detected, it is necessary to ask the information owner if he accepts or not to update the security policy. In Section VI we presented informations sent by the BPEL interpreter to the owner and the possible answers of the owner. To implement this functionality in OrchestraFlow we have decided to delegate to each owner to implement their own security service. This is a web service respecting a WSDL file describing the interface. This interface is common to all security services enabling OrchestraFlow to interact in the same way with all the security services. So when OrchestraFlow detects illegal flow of information, it makes a call to the web security service of the owner of that information (the address of the security service is sent with the security policy information at the beginning of the BPEL program execution). Figure 4 summarizes the interactions between services, the BPEL program and information owners.

VIII. Performances

To evaluate the performance of OrchestraFlow compared with the Orchestra BPEL interpreter, we tested the same BPEL programs on Orchestra and OrchestraFlow by varying the number and type of BPEL instructions in the programs, the number of initial data items and the size of labels as inputs.

The measurements of the execution times both with or without the information flow checking mechanisms were made using the nanoTime function of *Java*. We measure only the time of execution of the BPEL interpreter, without taking into account the communication times between the services (that are not constant from an invocation to another).

The results are presented in Table 2. They show an average overhead of 26%. This overhead takes into account the time involved by the label computation and propagation as well as the flow checking mechanisms. As the orchestrations used in our examples are pretty simple, the overhead is high. This is due to the fact that the additional security mechanisms are not negligible compared to the computations performed by the orchestrations. In the case of complex computations in the orchestration, we expect that the part of the security mechanisms would be much less important.

IX. Conclusion

The goal of our work is to give the user of a web service the ability to restrain the use of his data by services he never heard of. At the time of a service call, he is able to define which user data can be accessed by which web services. This property is guaranteed by a distributed security policy that defines which data can be accessed by which service. Using the security model defined by Myers et al. as a basis, our contribution consists in applying this type of security policy to Web Services and to dynamically define what are the variables in an orchestration of Web Services (written in a BPEL program) that are influenced by the user inputs. For this purpose, we follow the information flows that are produced by the various operations available in the BPEL interpreter. When flows are produced between variables, we update the labels attached to these variables to reflect the services that can read the data items. Thus, we can detect implicit or explicit data leakage and ensure the privacy of the user data. This approach proved to be feasible and lead to the implementation of the mechanisms inside the Orchestra BPEL interpreter.

However such an approach usually requires that the user knows all services involved in the orchestration. That is why we proposed a mechanism to dynamically update or build the security policy and principles for integrating this mechanism in OrchestraFlow. In particular, we defined a communication protocol between the BPEL program and the owner of the information.

Future work will focus on defining and implementing security services to ensure the authentication of actors and the confidentiality of the communications between the BPEL interpreter and the security service.

References

- [1] E. Cerami, *Web Services Essentials*. Sebastopol, CA, USA: O'Reilly and Associates, Inc., 2002.
- [2] OASIS, "Web services business process execution language version 2.0," OASIS Standard, April 2007. [Online]. Available: <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>
- [3] —, "Web services security: Soap message security 1.1," OASIS Standard Specification,, Feb 2006. [Online]. Available: <http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>
- [4] —, "Assertions and protocols for the oasis security assertion markup language (saml) v2.0," OASIS Standard, March 2005.
- [5] —, "extensible access control markup language (xacml) version 2.0," OASIS Standard, Feb 2005.
- [6] S. Nakajima, "Model-checking of safety and security aspects in web service flows," in *Web Engineering*, ser. Lecture Notes in Computer Science, S. B. . Heidelberg, Ed., vol. 3140/2004, 2004, p. 767.

- [7] D. Hutter and M. Volkamer, "Information flow control to secure dynamic web service composition," in *Security in Pervasive Computing*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006, vol. 3934.
- [8] S. Rossi and D. Macedonio, "Information flow security for service compositions," in *ICUMT*. IEEE, 2009, pp. 1–8.
- [9] K.-P. Fischer, U. Bleimann, W. Fuhrmann, and S. M. Furnell, "Security policy enforcement in bpel-defined collaborative business processes," in *ICDEW '07: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*. Washington, D-C, USA: IEEE Computer Society, 2007, pp. 685–694.
- [10] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *Proc. ACM Symp. on Operating System Principles*, pp. 129 – 142, October 1997.
- [11] A. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *IEEE Symposium on Security and Privacy*, 1998.
- [12] A. C. Myers, "Jflow: Practical mostly-static information flow control," *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pp. 228 – 241, 1999.
- [13] G. Hiet, V. Viet Triem Tong, L. Me, and B. Morin, "Policy-based intrusion detection in web applications by monitoring java information flows," *Int. J. Inf. Comput. Secur.*, vol. 3, no. 3/4, pp. 265–279, 2009.
- [14] V. Viet Triem Tong, A. Clark, and L. Mé, "Specifying and enforcing a fine-grained information flow policy: Model and experiments," in *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications*, 2010.
- [15] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, July 1977.
- [16] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003.

His main research interest focus on intrusion detection. He has published several national and international papers, and served on several conferences as program committee member.

Valrie Viet Triem Tong is associate professor at *École Supérieure d'Électricité (SUPELEC)*. Her favorite research topics mainly concern security and formal approaches in security.

Yves Le Traon is professor at Faculty of Science, Technology and Communication at University of Luxembourg, Campus Kirchberg, in the domain of software engineering, reliability, validation and security. He received his engineering degree and his PhD in Computer Science at the Institut National Polytechnique in Grenoble, France, in 1997. His research interests also include OO testing, design for testability, model-driven validation, model based testing, evolutionary algorithms and software measurement.

Author Biographies

Thomas Demongeot is engineer for the french Ministry of Defense since september 2008. He holds an engineering degree in computer science (2008) and is preparing a PhD. His main research interest focus on Service Oriented Architecture and information flow control.

Eric Totel is associate professor in Computer Science at *École Supérieure d'Électricité (SUPELEC)* since September 2002. He holds an engineering degree in computer science (1994), and a PhD in computer science (1998). He has been working on safety critical systems in space industry during four years, and is now associate professor since nine years.