Preventing data leakage in service orchestration

Thomas Demongeot, Eric Totel, Valérie Viet Triem Tong, Yves Le Traon

Preventing data leakage in service orchestration

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

Eric Totel
Valerie Viet Triem Tong
Supelec - Cesson-Sévigné - France
firstname.surname@supelec.fr

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

***Abstract*—Web Services are currently the base of a lot a e-commerce applications. Nevertheless, 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. 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 web services lead to situations where the policy prohibits the nominal operation of orchestration (e.g., when using a service that is unknown by the user). To solve this problem, we propose to let user to dynamically permit exceptional unauthorized flows. In order to make decision, the user is provided with all information necessary for decision-making. We also present an implementation inside the Orchestra BPEL interpreter. As far as we know this implementation is the first information flow monitor for web services that is also end-user configurable.**

I. Introduction

Web services 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 communication protocol level, but do not solve the problem of how to make a specific service orchestration trustworthy for the clients. Even if the service orchestration provider is trustworthy, it has no technical solution to guarantee 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, that expectations of the client must be expressed, which implies some security policy language is 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 checking at runtime whether the data protection policy is satisfied with a prototype tool called OrchestraFlow. This tool extends a BPEL (Business Process Execution Language) [1] interpretation engine, BPEL being the standard language for programming an orchestration of web services. BPEL is a relatively simple language that describes the sequences of service calls necessary to properly achieve a composite service. A BPEL program is a web service written in BPEL executed by a BPEL interpreter. A BPEL program can invoke other web services, thus it defines the orchestration of web services. BPEL program 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 users 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 present a state of the art on dynamic information flow tracking (Section II). We define a security policy that specifies legal information flows. We define what properties it can provide (Section III) and how to verify the policy (Section IV). Section V presents how to dynamically update the security policy. Finally we describe our implementation called OrchestraFlow (Section VI). Finally, we conclude and expose future work in Section VII.

II. Problem statement and related work

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 [2] aims at providing security for exchanging SOAP messages. Besides security architecture, there exist XML-based languages such as SAML (Security Assertion Markup Language) [3] and XACML (eXtensible Access Control Markup Language) [4] that allow specifying access control rules for accessing data or services. However there is no access control on the data once they have left their 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. 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.

In our approach, the security policy will be defined by users and can be updated at run-time, for instance when services are discovered. The security policy enforced at runtime relies on information flow tracking mechanisms that permit detecting user data leakage inside a BPEL interpreter.

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 [**?**] 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. [**?**] and [**?**] proposed type systems in order to guarantee non-interference property in dynamic service composition. But the method proposed by [**?**] needs to analyse each service involved in the orchestration and does not support complex orchestration. In [**?**] 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 [**?**], 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 [7] and [8] 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 [9] 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 [10], [11] 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 [11] can be either implemented at system level or at program level. In [10] 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 simply by changing the tags. In [11] the authors explain how to perform a modification of the policy by changing tag value but not 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 [10], [11] we explicitly identify user's data with numerical identifier. As Myers and Liskov in [7] and [8] 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.

III. Privacy Security Policy

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 user 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 [11] 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 and y are atomic information, $2 \times x, x + y, \ldots$ are compound information, the first non-atomic information results from the use of x, the second results from the use of x and y. Let us consider an e-shop service as a example of service. In this example, atomic information items are provided by the client : the chosen product, bank details and client email address. These atomic data items are used to compute all information items handled by the complete system, such as the total amount of the transactions, 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 sensitive information flows only into variables readable by authorized services. 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 statically (at the start of the service invocation) or dynamically (during the execution of the service orchestration) determining 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: $I = \{i_1, ..., i_n\}$ is the set of atomic information of the system. Information derived from several atomic information in $i_j, \ldots, i_k$ is denoted by $i_j \oplus \ldots \oplus i_k$

Services: $S = \{s_1, ..s_m\}$ 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., $i = i_j \oplus \ldots \oplus i_k$ then

$$owner(i) = owner(i_j) \cup \ldots \cup owner(i_k) \qquad (1)$$

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 $i = i_j \oplus \ldots \oplus i_k$ then

$$readers(i) = readers(i_j) \cap \ldots \cap readers(i_k) \qquad (2)$$

The security policy defines allowed readers for atomic information, rules of composition (1) and (2) define, by composition, readers of compound information. The policy is defined by the owners of information, since an owner determines the readers that are allowed to read its atomic information. A call to a service that brings 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 received information. The policy can be updated at any time by adding or removing a reader from the set of readers of information. An owner is responsible for removing readers from its own atomic information. When an information is compound, the several owners have to agree for any modification.

IV. Dynamic Checking of the Security Policy

In this work, the security policy is enforced through meta-data or simply labels put on every container of information: which means on every variable in a BPEL program. As it has been proposed by Myers in [7] 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 first modified when a new item is injected into the web service through a call to this service. At this moment, the injected information is considered 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 modifies 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 resulting flow 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. As stated before, a label is meta-data attached to each container and describes owners and readers of information currently located in the container. If c is a container its security label is of the form

$$L_c = \{\mathbf{i_1} : \mathbf{s}_\alpha \triangleright s_{\alpha_1}, ..., s_{\alpha_n}; ...; \mathbf{i_j} : \mathbf{s}_\beta \triangleright s_{\beta_1}, ..., s_{\beta_m}\}$$

Such a label means that information i contained in c is based on information $\mathbf{i_1}, ..., \mathbf{i_j}$. Information $\mathbf{i_1}$ is owned by $owners(\mathbf{i_1}) = \mathbf{s}_\alpha$ which authorizes readers $s_{\alpha_1}, ..., s_{\alpha_n}$. Depending on this label the readers allowed to access the information located in c are those authorized by all the owners, i.e., $readers(c) = \{\{s_{\alpha_1}, ..., s_{\alpha_n}\} \cap ... \cap \{s_{\beta_1}, ..., s_{\beta_m}\}\}$.
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.

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 $\{\mathbf{s_1} \triangleright s_{\alpha_1}, \ldots, s_{\alpha_n}\}$ where $s_{\alpha_1}, \ldots, s_{\alpha_n}$ are the readers of i

allowed by $\mathbf{s_1}$. 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 information flows and modify the content of information containers. 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 $c_j, ..., c_k$ labeled by $L_j, ..., L_k$ if we observe an information flow from the containers $c_j, ..., c_k$ to another container c, then we update the label of c which is now the union of labels attached to $c_j, ..., c_k$. Like Myers, we use the notation $L_j \sqcup \ldots \sqcup L_k$ 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 $c_j, ..., c_k$ 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 $c_j, ..., c_k$, i.e., the label should reflect the information history.

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

$$owners(L_1 \sqcup L_2) = owners(L_1) \cup owners(L_2)$$
$$readers(L_1 \sqcup L_2) = readers(L_1) \cap readers(L_2)$$
$$history(L_1 \sqcup L_2) = history(L_1) \cup history(L_2)$$

There is an example with three containers c_1, c_2 and c_3, respectively labeled by :

- $L_{c_1} : \{\mathbf{i_1} : \mathbf{s_1} \triangleright s_5, s_6; \mathbf{i_2} : \mathbf{s_1} \triangleright s_5, s_6\}$
- $L_{c_2} : \{\mathbf{i_1} : \mathbf{s_1} \triangleright s_5, s_6; \mathbf{i_3} : \mathbf{s_2} \triangleright s_6, s_7\}$
- $L_{c_3} : \{\mathbf{i_4} : \mathbf{s_3} \triangleright s_4, s_7\}$

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.

$\{\mathbf{i_1} : \mathbf{s_1} \triangleright s_5, s_6; \mathbf{i_2} : \mathbf{s_1} \triangleright s_5, s_6; \mathbf{i_3} : \mathbf{s_2} \triangleright s_6, s_7\}$

and means that

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

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.

V. Dynamic Update of the Security Policy

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

$$L_d = \{\mathbf{i_1} : \mathbf{s_1} \triangleright s_{1_1}, ..., s_{1_n}; ...; \mathbf{i_j} : \mathbf{s_j} \triangleright s_{j_1}, ..., s_{j_m}\}$$

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 $s \in \{s_{1_1}, ..., s_{1_n}, \ldots s_{j_1}, \ldots s_{j_m}\}$. If s is an authorized reader then the BPEL program performs the call. Otherwise we ask owners of s to confirm if the call must be authorized anyway. 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 $(\mathbf{s_1}, \ldots, \mathbf{s_j})$ if they authorize or not sending a compound information d computed using their atomic information resp. $\mathbf{i_1}, \ldots, \mathbf{i_j}$.

More precisely the BPEL interpreter calls a dedicated service to contact the information owners. 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 sent to/by this service. This service is used to ask every owner $\mathbf{s_k}$ 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 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 owners accept the modification, the label is modified: s is added as reader for d.

VI. OrchestraFlow : a dynamic monitor for BPEL

In this section we present OrchestraFlow which implements the model detailed in the previous sections as a patch for the BPEL interpreter Orchestra. OrchestraFlow taints variables

of a BPEL program using labels as detailed before, the implementation of labels is presented in this section. 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.

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 order to allow dynamic updates 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 compatible with existing BPEL interpreters that do not carry out our protection mechanisms.

In OrchestraFlow, a label is therefore 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.

Each variable 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.

The label of a variable is updated at each observation of an information flow. As defined early by D. Denning in [12], *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.* Here we distinguish *implicit or explicit information flow*. *An implicit information flow signals information through the control structure of a program* [13]. The reader will find a complete survey on this subject in [13]. First we focus on explicit information flows between variables which are transfers of information induced by operations made by the program involving these variables. Among them `Assign, Invoke, Receive, Reply` induce explicit information flows. OrchestraFlow extends Orchestra in order to update concerned labels at each call of one of the mentioned operations.

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 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 assignement 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 information contained in the XPath expression according to the definition IV.

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 sent messages (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 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 IV.

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

The second type of information flow that can be created by the language is of implicit type. It is what happens for example during conditional operations and loops. In these cases, data manipulated within the structure of the loop or conditional depend 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 a flow from c to x. The label of x is computed from the labels of e and c according to the definition IV.
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. 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 these functions is called, we verify that the service call complies with the security policy, i.e., 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 $s \in reader(m)$ as defined in definition IV. 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.

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 V we presented information sent by the BPEL interpreter to the owner and the possible answers of the owner. To implement this functionality in OrchestraFlow we 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 an illegal information flow, 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).

VII. 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 was 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 security mechanism in order to allow updates of an information flow security policy by the user of a BPEL program.

References

[1] 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

[2] ——, "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

[3] ——, "Assertions and protocols for the oasis security assertion markup language (saml) v2.0," OASIS Standard, March 2005.

[4] ——, "extensible access control markup language (xacml) version 2.0," OASIS Standard, Feb 2005.

[5] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[6] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 85–96, 2004.

[7] 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.

[8] A. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *IEEE Symposium on Security and Privacy*, 1998.

[9] A. C. Myers, "Jflow: Pratical mostly-static information flow control," *Proceedings of the 26th ACM Symposium on Principles of Programming Langages*, pp. 228 – 241, 1999.

[10] 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.

[11] 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.

[12] 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.

[13] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003.