Information Flow Control on a Multi-Paradigm Web Application for SQL Injection Prevention

Meriam Ben-Ghorbel-Talbi, François Lesueur, and Gaetan Perrin

Université de Lyon, CNRS
INSA-Lyon, LIRIS, UMR5205, F-69621, France
{meriam.talbi, francois.lesueur, gaetan.perrin}@insa-lyon.fr

Abstract. In this paper, we propose an integrated framework to control information flows in order to prevent security attacks, namely, SQL injections threatening data confidentiality. This framework is based on the Prerequisite TBAC model, a new Tuple-Based Access Control model designed to control data dissemination in databases, and that guarantees a controlled declassification. To track information flow in the application part, we propose to propagate dynamically security labels through the system using Paragon, a typed-security language that extends Java with information flow policy specification.

Keywords: Information Flow, TBAC, Declassification

1 Introduction

When trying to control information flows in a program, most propositions focus on only one part: either security-typed languages for imperative programs or Multi-Level Security databases for the declarative parts. Although programs are in fact constructed using both imperative and declarative programming, few previous work study both at the same time. In this paper, our contribution is two-fold: we propose an integrated framework to follow information flows from the moment they enter the system until they leave it, possibly being stored and manipulated in the database in the meantime. Moreover, this framework is based on Dissemination Control to circumvent the threats of uncontrolled declassification. We argue that the combination of these contributions allows to greatly reduce the burden of the application developer: the end-to-end aspect allows to dynamically tag data entering the system (proxy service) rather than variables in the code and then to control output only when it leaves the system, the application part can be mostly unchanged. The dynamic aspect allows the developer not to specify security labels on its variables, inside his code, but in the database at the tuple level. Dissemination control prevents erroneous declassifications, since data entering the system are tagged with their allowed ways of being declassified and the developer can declassify according to these tags without worrying of a confidentiality breach. Hence, our framework can be used to prevent information leakage, such as SQL injections.

In this paper, we propose to use TBAC [1], a new Tuple-Based Access Control model designed to control the information flow in databases. The objective of TBAC is to provide a mechanism that controls the dissemination of tuples according to the authorizations defined by their producers. It is designed in the same spirit as the decentralized information models [2] in the sense that users are allowed to specify their own security policy on their data. To deal with declassification we propose to extend the TBAC model family by a new instance, which we call *Prerequisite* TBAC, that provides means to define which conditions have to be satisfied to declassify data and by which subjects. In order to control the dissemination throughout the system, namely in the application part, we propose to use Paragon [3], a typed-security language that extends Java by adding the ability to label data information flow policies. Paragon is well adapted to our requirements and provides expressive information-flow policies to deal with the declassification issue. Moreover, it supports runtime policies which is an important feature to implement the dynamic aspect of our approach.

2 Related work

Several works in the literature have proposed information flow control solutions based on the multilevel security policy model (MLS). Work in databases have proposed MLS DBMS to enforce information flow control [4–6]. In these models objects are passive entities such as relations, tuples, or rows. Subjects are active entities such as users or programs. Many MLS database systems have been also proposed, such as Oracle Label Security (OLS) [7], PostgreSQL [8], Sybase Secure SQL Server [9]. Work in programming languages, such as FlowCaml [10], have addressed information-flow control by proposing security-typed languages. They use tainting mechanisms by labeling variables as tainted or untainted in order to control programs inputs and outputs. The basic concept of these languages is to statically analyze the source code of a program at compile time in order to check that all the performed operations respect the security policy.

Recently, more research has focused on the Decentralized Label Model (DLM) to deal with decentralized systems requirements. An application of the DLM model to programming languages was proposed in [11] called Jif, for Java Information Flow. Jif extends Java by adding labels that express restrictions on how information may be used. Paragon [3] is another security-typed extension to Java, which builds on the recent policy language Paralocks proposed in [12]. The main strength of Paragon over Jif is that it is more general in the sense that the DLM policy lattice is a sub-lattice of Paralocks.

However, few work have addressed the end-to-end information flow control issue : to our best knowledge, the IFDB model [13] is the only one that proposes to track flows and enforce security policy both in the DBMS and the application platform. It introduces the query by label concept: each query has an associated label, which is the label of the process issuing the query. This label is used as a filter. Authority is bound to principals, such as users and roles, and each process runs with the authority of a particular principal. Authority can be delegated

and given to users, application procedures, and also to stored procedures and views in order to allow declassification. Other work have been proposed in this same optics, and have pointed out the need to deal with a uniform information flow control between databases and applications. As mentioned in [14], most common web attacks are attacks across component boundaries (e.g. injection attacks, cross-site scripting attacks). In [15] authors have designed DBTaint system. It extends the database to associate each piece of data with a taint tag and propagates these tags during database operations. In [16], information-flow policies are specified in the database query interfaces and enforced in the web scripting language by a static type checker. In [17], authors present LabelFlow, an extension of PHP that tracks the propagation of information dynamically throughout the application, transparently both in the PHP runtime and through the database. IFDB is the most close to our approach as it is based on the DLM model. But, contrary to the IFDB model which tracks flows on a per-process granularity in the application-side, we aim to deal with a fine-grained flow control both in the DBMS and the application platform as in [15–17]. Moreover, we aim to deal with the declassification in the security policy in order to specify how data can be declassified, which is not the case in these previous work.

3 System Architecture

As shown is figure 1, the security policy is sticked to data as labels (called *s-tags*), all throughout the framework, so that data are filtered when they leave the entire system and not only the database. Moreover, we specify the declassification policy in the label itself, thus data can be declassified according to their labels, which guarantees a controlled declassification.

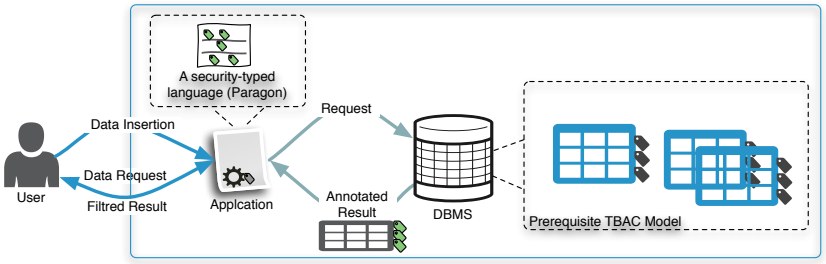

Fig. 1. System Architecture

3.1 The Security Model

We propose to deal with the security policy using a new instance of the TBAC model [1], which we call the *Prerequisite* TBAC. This new instance provides facilities to control the dissemination of data based on the policy attached to them and also allows to express some form of authorized declassification. In this model, we consider that a user is allowed to access a data if and only if the

prerequisites expressed by the data owners have been previously satisfied. These prerequisites are linked to data and express which treatments these data must go through and by which subjects. Formally, we consider that Pre is the set of expressible prerequisites, U is the set of users, and each tuple t is annotated by an *s-tag* t_{auth}. An *s-tag* is a disjunction of atomic $tags$, each atomic *tag* being defined as $((p, U_v), U_r)$, meaning that someone in U_r can read if someone in U_v validate all the prerequisites in p, where $p \subseteq Pre$, $U_v \subseteq U$, $U_r \subseteq U$. An empty prerequisite means that users in U_r can access this tuple without any prerequisite.

3.2 The Application-side

To track information flow in the application-side, we use Paragon which builds on two basic components: actors and parameterized locks. Actors can be principals or specific communication channels. Locks are a boolean variables used to communicate the security relevant state of the program to the policy. A policy is composed of a set of clauses and each clause must have a head specifying to which actors the information may flow, and may have a body that specifies in which conditions data may flow to these actors. Note that, Paragon policies are similar to our policy definition: the prerequisite conditions and Locks have the same semantic and are both used to specify how to declassify data.

Runtime policy. After receiving data from the database, we instantiate the policies of the variables in the application code using *s-tags* that are attached to the query result. Hence, *s-tags* will be propagated from the database to the application and they will be sticked to data until they leave the entire system.

Downgrading. In our model, prerequisite conditions are specified using locks in the application-side. According to the system state, or after some data transformation that validates the prerequisite conditions, locks will be opened and hence data will be declassified. As we said previously, prerequisite conditions have to be removed from *s-tags* as soon as they are validated. Thus, we have to define a *downgrade* function in order to re-annotate policies by deleting opened locks. We call this function $explicit\ declassification$.

Filtering. In Paragon, Input-Output channels are actors so that they can be labeled with a security policy. Thus, to automatically control data that flow to the user, we just need to tag Output channels with a policy containing the current user's credentials as authorized actor. Hence, only data that satisfy the security policy will flow from the application to the user.

3.3 The Database side

The TBAC model has been defined in [1] where different instances have been proposed. The TBAC models family evolves around the propagation and combination of access rights on tuples to provide information flow control in a relational ecosystem. In this paper, we propose a new instance, called *Prerequisite* TBAC, to deal with the declassification policy.

Propagation TBAC uses the provenance framework described in [18] to propagate and combine *s-tags*. In databases, SPJRU queries are used for computations and are the place for access rights propagation and combination. *Select* and *Rename* are transparent as they do not alter the set of *s-tags*. *Project* and *Union* can merge several original tuples into the same one. Each tuple t in the result can be equivalently derived from a set T of tuples (two in the case of $\cup$) which have to be combined additively, thus, access to t should be granted if access is granted to at least one tuple from T. *Join* combines two original tuples into a composite one: access to a joined tuple needs access rights to all the original ones.

We can restate the SPJRU semantics from an access control point of view informally by *one may access to a piece of information if he is authorized to access to the original tuples which contribute to it.* More formally, if we consider two tuples a and b, where $a_{auth} = \{tag_{a_1} \vee \ldots \vee tag_{a_n}\}$ and $b_{auth} = \{tag_{b_1} \vee \ldots \vee tag_{b_m}\}$, annotations are combined with relational queries as follows :

- If $t = a \bowtie b$, access to t requires access to both a and b. Then, t's annotation is defined as a disjunction of a conjunction of atomic tags as follows: $t_{auth} = \{(tag_{a_1} \wedge tag_{b_1}) \vee \ldots (tag_{a_1} \wedge tag_{b_m}) \vee \ldots (tag_{a_n} \wedge tag_{b_m})\}$. For each conjunction of atomic tags we have: $((p_{a_i}, U_{va_i}), U_{ra_i}) \wedge ((p_{b_j}, U_{vb_j}), U_{rb_j}) = \{((p, U_v), U_r) | p = p_{a_i} \cup p_{b_j}, U_v = U_{va_i} \cap U_{vb_j}, U_r = U_{ra_i} \cap U_{rb_j}\}$.
- If $t = a \cup b$, access to t requires access to any of a and b and t's annotation is defined by $t_{auth} = \{tag_{a_1} \vee \ldots \vee tag_{a_n} \vee tag_{b_1} \vee \ldots \vee tag_{b_m}\}$. Simplification must be applied for tags having the same prerequisite sets as follows: $\{((p, U_v), U_r) | U_v = U_{va_i} \cup U_{vb_j}, U_r = U_{ra_i} \cup U_{rb_j}\}$.

Implicit Declassification is used when the query validates automatically the prerequisite condition. For instance, if a prerequisite requires the data to be aggregated and if the query is an aggregation then the declassification can be automatically triggered. If the current user is allowed to validate this prerequisite, then it is replaced by an empty one, allowing users in the second part of the rule to access this tuple. We consider that, a user u is authorized to validate the prerequisite when either:

- u is expressed in the first part of the rule as a prerequisite user or,
- the prerequisite user specified in the first part of the rule is equal to *all* (i.e. every user is authorized to validate the prerequisite).

Pre-Filtering is used in order to optimize the query result by sending only data that can be accessed or declassified by the current user in the application-side. The pre-filtering function f is applied on every tuple composing the result to a query. For a user u requesting a tuple t, f returns *true* if and only if:

- u is authorized to validate the prerequisite conditions, which means that explicit declassification is required in the application-side in order to access data,
- u is expressed in the second part of the rule as an authorized user with no invalidated prerequisites left.

4 Implementation

We present here a first implementation attempt of our approach. As shown in figure 2, we have three components: the database, implemented using HSQLDB (HyperSQL DataBase), that stores data and *s-tags* at the tuple-level; the application, implemented in Java, that interacts with the database to request and insert data; and the proxy, implemented in Paragon, that controls I/O channels, by adding *s-tags* before inserting data in the database, and by filtering data before they leave the application.

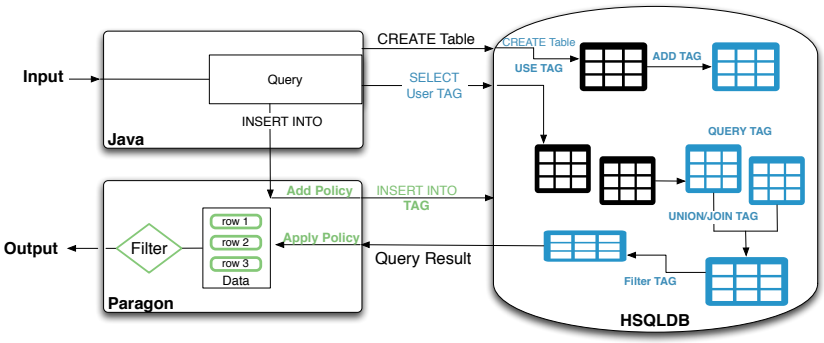

Fig. 2. The implementation architecture

Modifications in the Database. HSQLDB is a relational database software written in Java. We have extended it to deal with *s-tags*. For this purpose, we have implemented a custom SQL parser that modifies all SQL queries at runtime:

- When tables are created, we intercept the query to add the *USETAG* command that adds automatically a new column called *STAG*. This column is used to store the security policy at the tuple level.
- When data are requested, we first add the *UserTAG* command to the SQL query to specify the current user credentials. Then, we intercept the query result and we run our algorithm that combines *s-tags* according to the SQL query. After the *s-tags* calculation, we run our pre-filtering function that decides which tuples can be sent to the application-side, according to the current user credentials.

The Application. We have developed a basic application in Java that plays the role of an interface between the user and the database in order to insert and request data. This application is a ToDo list, where users can signin, create, show one or all their tasks. A given task can be assigned to one or many users, and users can specify their security preferences in order to share their tasks or to allow other users to see some details about them (e.g. the title, or the number of their tasks).

The Proxy. Paragon is used to develop the proxy that controls input/output channels. Using *.pi* files that tie paragon files (*.para*) with java program, it is possible to specify the policy annotations only in the application part where data flows from the application to the database and inversely. To control input channels, entering data are intercepted before inserting them to the database. Hence, the INSERT command is modified in order to add *s-tags* that are derived according to the security policy specified in the application-side. We currently use a default policy: users are only allowed to access to tasks that are assigned to them. To control output channels, the proxy has to dynamically instantiate the policy of the variables constituting the query result by their *s-tags*. Thus, for each tuple in the query result, we convert its corresponding *s-tag* to a paragon policy and we assign it to data. In addition, a new policy is created using the current user credentials and assigned to the output channel. Data are, hence, filtered accordingly. This is possible using the Paragon runtime library that allows to support dynamic features.

Testing SQLI Attacks. SQL Injection Attacks are used by attackers in order to violate data confidentiality and integrity. They use different techniques to modify or inject an SQL query in the application input that is sent and executed in the database. To validate our approach, we have tested some SQLI attacks and the query result was successfully blocked by our proxy. As we said earlier, in our model, we only deal with attacks threatening data confidentiality. For instance, in our application, a user, say *Bob*, can use an SQLI attack in order to show all tasks that are stored in table $Tasks$. Even, if the SQLI succeeds in the database-side and the whole table is returned, the output result shown to *Bob* is filtered and only tuples having *Bob* as authorized user are kept, namely only tasks assigned to *Bob*. Obviously *Bob* will see all fields composing the tuple, even if he is actually not supposed to see them. It is our design choice to use a tuple granularity instead of labeling fields, but, this can be an interesting issue to be investigated as future work.

5 Conclusion

Our framework provides end-to-end security guarantees on declassification, by specifying in the labels themselves how data can be declassified and by which users. This is an important feature that allows to greatly reduce the burden of the application developer and thus to prevent security attacks. As we said in this paper, few existing works have addressed the end-to-end information flow control, and they do not deal with controlled declassification as it is the case in our work. We have focused on attacks threatening data confidentiality and we have proposed a proof of concept implementation to demonstrate that our approach is feasible. Note that, our aim was to let the application part as unchanged as possible, in order to facilitate the integration of our approach in existing programs. For this purpose, we have proposed: a custom SQL parser that modifies SQL queries at runtime to add and combine *s-tags*, resp. when

data are inserted and requested from the database; and a proxy service that is in charge to label data entering the system, to dynamically propagate the *s-tags* from the database to the application, and then to control output when data leave the system. Currently we are working on extending our prototype to deal with all the features of our model, namely the implicit and explicit declassification, in the database and the proxy-side, respectively. As future work, we aim to design a complete prototype and to evaluate its performance.

Acknowledgments. This work has been partially funded by the French ANR KISS project under grant No. ANR-11-INSE-0005.

References

1. Thion, R., Lesueur, F., Talbi, M.: Tuple-Based Access Control: a Provenance-Based Information Flow Control for Relational Data. In: SEC@SAC. (2015)
2. C., M.A., Liskov, B.: A Decentralized Model for Information Flow Control. In: SOSP. (1997)
3. Broberg, N., van Delft, B., Sands, D.: Paragon for Practical Flow-Oriented Programming. In: APLAS. (2013)
4. Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R.: The SeaView Security Model. IEEE Transactions On Software Engineering **16**(6) (1990)
5. Sandhu, R., Chen, F.: The Multilevel Relational Data Model. ACM Transactions on Information and System Security (1998)
6. Smith, K., Winslett, M.: Entity modeling in the MLS relational model. In: VLDB. (1992)
7. Jeloka, S.: Oracle Label Security Administrator' s Guide, 11g Release 2 (11.2). Technical report, ORACLE (2013)
8. PostgreSQL Global Development Group: PostgreSQL 9.1 Documentation (2011)
9. Sybase Inc. Building Applications for Secure SQL Server: Sybase Secure SQL Server Release 10.0. Technical report (1993)
10. Simonet, V.: FlowCaml in a nutshell. In: Proceedings of the first APPSEM-II Workshop. (2003)
11. MYERS, A.C.: JFlow: Practical mostly-static information flow control. In: POPL. (1999)
12. Broberg, N., Sands, D.: Paralocks – Role-Based Information Flow Control and Beyond. In: POPL. (2010)
13. Schultz, D., Liskov, B.: IFDB: Decentralized Information Flow Control for Databases. In: CCS. (2013)
14. Schoepe, D., Hedin, D., Sabelfeld, A.: SeLINQ: Tracking Information across Application-Database Boundaries. In: ICFP. (2014)
15. Davis, B., Chen, H.: DBTaint: Cross-application Information Flow Tracking via Databases. In: WebApps. (2010)
16. Peng, L., Zdancewic, S.: Practical Information Flow Control in Web-based Information Systems. In: CSFW. (2005)
17. Chinis, G., Pratikakis, P., Athanasopoulos, E., Ioannidis, S.: Practical Information Flow for Legacy Web Applications. In: ICOOOLPS, ACM (2013)
18. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance Semirings. In: Proceeding of the 26th symposium on Principles Of Database Systems (PODS). (2007)