Formal Analysis of Web Applications with Insecure Database APIs due to SQL Injection Attacks (Extended Abstract)

Federico De Meo[1], Marco Rocchetto[1], and Luca Viganò[1,2]

[1] Dipartimento di Informatica, Università di Verona, Verona, Italy
[2] Department of Informatics, King's College London, London, UK

Context and motivation. Many programming languages (such as PHP, Java, Python) define a set of APIs that provide database functionalities that support developers in implementing the interaction between a web application and a database. These APIs provide functionalities like opening or closing a connection to a database or performing raw SQL queries. Generally speaking, APIs for querying databases do not always provide a secure defense mechanism against the most successful attack [6, 2] to web applications, namely, *SQL-injection (SQLi)*. For example, the PHP language offers three different APIs to connect to MySQL: mysql, mysqli and PDO. All of these APIs provide functions to execute raw SQL queries but none of these functions provides any security check.

The widespread use of such APIs thus constitutes a serious threat to the security of web applications, which are rapidly becoming one of the most important resources of the web. Due to the complexity of modern web applications, SQLi can be difficult to detect and experience has shown that manual analysis of web applications is a highly error-prone activity. It is also well known (e.g., [4]) that state-of-the-art vulnerability scanners do not detect vulnerabilities linked to logical flaws of web applications. This means that even if a vulnerability scanner or a specific tool for the detection of SQLi (e.g., sqlmap) can concretely discover a SQLi, it cannot link SQLi to logical flows that lead to a violation of a generic security property, e.g., the confidentiality of a data accessible only bypassing an authentication phase with a SQLi.

A number of formal approaches for the analysis of web applications, based on the *Dolev-Yao (DY) intruder model* [3], have been implemented (e.g., [1, 8] to name just two that we have helped develop). However, the DY model typically reasons about cryptographic operators and their algebraic properties (e.g., operators such asymmetric and symmetric cryptography, or modular exponentiation) and about how such operators are used to encrypt messages, but abstracts away the contents of the payloads of the messages. As a consequence, these approaches cannot properly detect and exploit SQLi.

Contribution. Our contribution is the definition of a formal approach for the representation of SQLi and attacks that exploit SQLi in order to violate security properties of web applications. We define how to formally represent a web application that interacts with a database using insecure APIs and we propose an extension of the DY model that can deal with SQLi. It is important to point out that we do not find SQLi attacks, but rather we exploit SQLi attacks in order to analyze security properties of web applications that make use of database APIs.

In order to explain the details of our approach, we need first to explain which type of SQLi we are considering. Some attempts at giving general classifications of SQLi have already been considered in [5, 7]. Based on these, we divide SQLi into five different categories: (i) boolean-based blind, (ii) time-based, (iii) error-based, (iv) union query, (v) stacked queries, and (vi) second-order. For concreteness (and brevity), in this paper, we focus on *Boolean-Based Blind*

SQLi (*BBB*), but we have been working at the formalization of an intruder model that can cope with all the aforementioned categories and the resulting attacks.

A BBB is an injection in which an intruder inserts into an HTTP parameter, which is used by a web application to write a SQL query, one or more valid SQL statements that make the WHERE clause of the SQL query evaluate to true or false. By interacting with the web application and comparing the responses, the intruder can understand whether or not the injection was successful. In this way, an intruder can perform two powerful attacks:

- *authentication attack*: the intruder bypasses an authentication check,
- *data extraction attack*: the intruder extracts data from the database.

Our approach can currently handle only authentication attacks but we have also been working on the case of data extraction attacks. In the remainder of this paper, we will thus focus only on authentication attacks. In such an attack, the intruder injects a statement (into a parameter used by the web application to create a SQL query) that changes the truth value of a WHERE clause in a SQL SELECT query, creating a tautology. If a web application performs an authentication check querying a database, this attack will then trick the database into replying in an affirmative way even when no authentication, or an incorrect username and password pair, have been presented by the intruder. This implies that if we want to reason about the consequences of this kind of SQLi in a formal specification of a web application, we need to formalize: (i) the behavior of the client interacting with the web application, (ii) the behavior of the web application (and its interaction with the database) and (iii) the behavior of the database as well as the evaluation of WHERE clauses.

A concrete example. We now illustrate our approach by means of a concrete example. In the MSC (Message Sequence Chart) in Figure 1, there are three entities: a client, a server hosting the web application and a database with which the server has a long lasting relation, i.e., no intruder can read or modify the communication between the two. First, the client sends a message to the server, containing his username and password (1). This is the standard behavior of most web applications: a client via a browser fills in the login form and clicks on a submit button to send such data to the server hosting the web application.[1] The web application then creates a SQL query and (using some API) sends it to the database (2). We only show the instantiation of the WHERE clause in the SQL query with `Username` and `Password` representing the username and password columns in the database, respectively. The database checks the SQL query (3) it has received and replies to the server with a variable `Response` (4) that represents login acceptance in case the SQL check in the database returns one or more tuples, refuse otherwise. The server finally forwards the response to the client (5).

We have used the formal specification language ASLan++ [9] to write a specification of all these entities (mainly because ASLan++ handles databases as shared sets of tuples). Roughly speaking, in ASLan++, we represent the client and server entities as separate parallel programs that communicate with each other. The database is a sub-entity of the server.

Web application. The `Client` entity sends a message and receives back a response. He instantiates the two usual login fields (represented by the variables `Username` and `Password`) and sends them to the `Server`.

[1] We omit here any previous communications between the client and the server because these are not useful for the attacks we want to show. However, in our approach, we could of course also consider these messages.

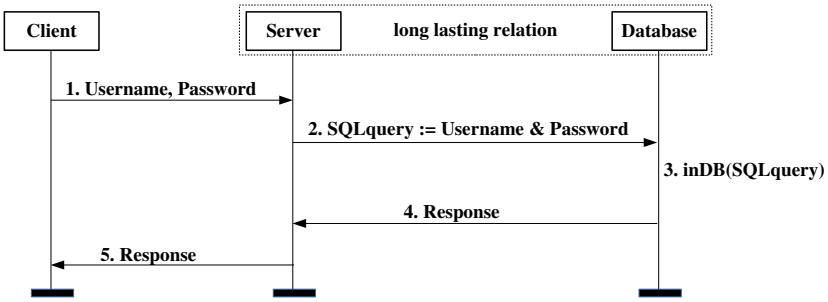

Figure 1: MSC of an authentication phase in a web application

```
Client{
        Username := alice;
        Password := pass;
        Client -> Server: Username.Password;
        Server -> Client: Response;
}
```

The `Server` entity receives the message from the `Client`, creates a SQL query and sends it to the database. Once the server receives back a response, it forwards it to the client.

```
Server{
        Client -> Server: Username.Password;
        SQLquery:=loginTable.Username.Password;
        Server -> Database: SQLquery;
        Database -> Server: Response;
        Server -> Client: Response;
}
```

Abstracting away some detail, the SQL query is only composed by a table (`loginTable` in our case) and a concatenation of formulas. `Username` and `Password` variables represent two equalities between username and password fields of the database and the instantiation of the two variables, i.e., `alice` and `pass`, respectively.

Once the `Database` entity receives a query from the server, it checks the `WhereClause` in the table `loginTable` and answers with a constant `dbtuple` only if the check is successful (the `inDB` predicate behavior is described by Horn Clauses and explained later in this section). This check represents the search inside of the database table `loginTable` of a row with columns `Username` and `Password` instantiated to `alice` and `pass`, respectively.

```
Database{
  if(Server -> Database: Table.WhereClause & inDB(WhereClause,Table)){
  Database -> Server:  dbtuple;
}}
```

Goal. The goal for our web intruder is to find a way to obtain the constant `dbtuple` without knowing the correct credentials, i.e., a violation of authentication and confidentiality.

Web intruder. We now briefly introduce our extension of the classical DY intruder model in order to extend the intruder's ability to reason about SQLi. Given that we only deal with

BBB, we extend the DY intruder by giving him the ability to send a concatenation of boolean formulas made of conjunctions and disjunctions in order to create the actual injection that modifies the evaluation of a WHERE clause to true. This characteristic highlights an important difference between the classical DY intruder and the enhanced version we are proposing: our web intruder works with payload rather than messages. This means that we are actually interested in modeling the structure of the message the intruder is sending to the web application. For example, contrast the abstract message $\{M\}_K$ sent by the standard DY intruder, where some message M is encrypted with a key K, with $\{M_1 \vee M_2\}_K$ where the form of the message is specified: what is encrypted is the disjunction of two sub-messages (one of which, in the case of a BBB, trivially evaluates to true, e.g., M_2 is "$1 = 1$").

Experiments and results. We have performed a number of preliminary experiments using the AVANTSSAR Platform [1], which includes the three model checkers CL-Atse, OFMC and SATMC. Since none of these tools allow for modifications to the DY intruder, we have represented conjunctions and disjunctions as Horn clauses and we have added these to the specification of the web application. These Horn clauses were used to describe the behavior of the database (`inDB` predicate) when evaluating a WHERE clause. This description, together with the ability of a DY intruder of exploring all possible message concatenations, should indirectly give to the intruder the possibility of injecting boolean formulas. However, when we first ran our model, we encountered a non-termination problem even with the simple example of this paper. We have then tried to simplify our model. The simplification comes from the fact that any boolean formula made of disjunctions and conjunctions can always been made to evaluate to true by appending the sequence *OR TRUE*. We are not providing a full proof here, but give only a brief proof sketch: given that WHERE clauses only use the two connectives AND and OR and that what we want to achieve is to modify a WHERE clause evaluation to true, if we consider the truth table of the formula (in propositional logic restricted to the two connectives *OR* and *AND*) *A OR B*, then *A OR B* evaluates to true for any *A* whenever *B* is true.

We thus have changed the ability of our intruder by allowing him to concatenate the exact payload, OR TRUE, and defined a Horn clause to model that whenever a formula has OR TRUE injected by the intruder, that formula evaluates to true. This has allowed the AVANTSSAR platform to terminate in a few milliseconds, without losing the possibility of performing a BBB and successfully finding an SQLi attack bypassing a login phase. Running the AVANTSSAR platform against our specification, we obtained the expected attack trace in which the intruder `i` bypasses the login phase:

```
1 i        -> server   : Username.or.true
2 server   -> database : loginTable.Username.or.true
3 database -> server   : dbtuple
4 server   -> i        : dbtuple
```

In the first message, `i` sends to the server a variable `Username` (its instantiation is not important) and, instead of sending a password, he instantiates the variable `Password` with `or.true` where `true` represents any true statement understood by a database, e.g., $1 = 1$. The server sends to the database the SQL query that is now a tautology, so that the intruder obtains `dbtuble`, thus violating its confidentiality as well as authentication.

Conclusions. We have been developing a formal approach to analyze a web application that interacts with a database using insecure APIs (which required us to define an extension of the DY model that can deal with SQLi). At this stage, our work deals only with BBB that leads

to authentication attacks, but we are already working at extending to data extraction and to include all other possible SQLi categories.

References

[1] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, SerenaElisa Ponta, Marco Rocchetto, Michael Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *Tools and Algorithms for the Construction and Analysis of System*, LNCS 7214, pages 267–282. Springer, 2012.

[2] Steve Christey. The 2009 CWE/SANS top 25 most dangerous programming errors. `http://cwe.mitre.org/top25`.

[3] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[4] Adam Doupé, Marco Cova, and Giovanni Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS 6201, pages 111–131. Springer, 2010.

[5] William G. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *IEEE International Symposium on Secure Software Engineering*, pages 65–81. IEEE, 2006.

[6] OWASP. *OWASP Top 10 for 2013*. OWASP, 2013. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[7] Amirmohammad Sadeghian, Mazdak Zamani, and Shahidan M. Abdullah. A taxonomy of sql injection attacks. In *International Conference on Informatics and Creative Multimedia*, pages 269–273, 2013.

[8] Luca Viganò. The spacios project: Secure provision and consumption in the internet of services. In *Software Testing, Verification and Validation*, pages 497–498, 2013.

[9] David von Oheimb and Sebastian Mödersheim. ASLan++ — a formal security specification language for distributed systems. In *Formal Methods for Components and Objects, FMCO*, LNCS 6957, pages 1–22. Springer, 2010.