Formal Analysis of Vulnerabilities of Web Applications Based on SQL Injection*

Federico De Meo[1], Marco Rocchetto[2], and Luca Viganò[3]

[1] Dipartimento di Informatica, Università degli Studi di Verona, Italy
[2] iTrust, Singapore University of Technology and Design, Singapore
[3] Department of Informatics, King's College London, UK

Abstract. We present a formal approach for the analysis of attacks that exploit SQLi to violate security properties of web applications. We give a formal representation of web applications and databases, and show that our formalization effectively exploits SQLi attacks. We implemented our approach in a prototype tool called SQLfast and we show its efficiency on four real-world case studies, including the discovery of an attack on Joomla! that no other tool can find.

1 Introduction

Motivations. According to OWASP (the Open Web Applications Security Project [27]), *SQL injection (SQLi)* is the most critical threat for the security of web applications (*web apps*, for short), and MITRE lists improper SQLi neutralization as the most dangerous programming error [6]. SQLi was first defined in [14] but, also due to the increasing complexity of web apps, SQLis can still be very difficult to detect, especially by manual *penetration testing* (*pentesting*).

A number of SQLi scanners have thus been developed to search for injection points and payloads, most notably *sqlmap* [33], which allows human pentesters to find SQLi vulnerabilities by testing the web app with different payloads, and *sqlninja* [34], which focuses on SQL server databases. The combination of the two provides the pentester with a powerful tool suite for SQLi detection. However, neither sqlmap nor sqlninjia (nor other state-of-the-art vulnerability scanners) are able to detect vulnerabilities linked to logical flaws of web apps [12]. This means that even if a scanner can concretely discover a SQLi, it can't link SQLi to logical flaws that lead to the violation of a generic security property, e.g., the secrecy of data accessible only bypassing an authentication phase via a SQLi.

Moreover, determining that a web app is vulnerable to SQLi (and which payload to exploit) might not be enough for the app's overall security. Consider, for instance, a web app that relies on legacy code (when an update is not feasible, e.g., because the legacy code is a core part of the system). If a SQLi is found, an investigation should be performed to understand when the SQLi can be exploited and whether this compromises security. This investigation is carried out manually by the pentester in charge of identifying attack scenarios, thus potentially leading to additional omissions, errors and oversights in the security analysis.

* This work was carried out while Marco Rocchetto was at the Università di Verona.

A number of formal approaches for the security analysis of web apps, based on the *Dolev-Yao (DY) intruder model* [11], have been implemented recently, e.g., [1,3,4,31,36]. However, the DY model is typically used to reason about security protocols and the cryptographic operators they employ (e.g., for asymmetric or symmetric cryptography, modular exponentiation or exclusive-or) but abstracting away the contents of the payloads of the messages. As a consequence, these approaches cannot properly identify or exploit new SQLi payloads since reasoning about the contents of the messages is crucial to that end.

Contributions. In this paper, we present a formal approach for the analysis of attacks that exploit SQLi to violate security properties of web apps. We define how to formally represent web apps that interact with a database and how the DY intruder model can be extended to deal with SQLi.

In order to show that our formalization can effectively be used to detect security vulnerabilities linked to SQLi attacks, we have developed a prototype tool called *SQLfast (SQL Formal AnalysiS Tool)* and we show its efficiency by discussing four real-world case studies. Most notably, we use SQLfast to detect an attack on Joomla! which, to the best of our knowledge, no state-of-the-art SQLi scanner (e.g., sqlmap or sqlninja) can detect since they do not automatically link different attacks in one attack trace (i.e., they do not find logical flaws linked to SQLi attacks). Another key novel aspect of SQLfast is that it can detect complex attacks in which a first SQLi attack provides data for a second subsequent attack. We show that SQLfast allows us to exploit SQLi combining it with logical flaws of web apps to report sophisticated attack traces in a few seconds and can also deal with Second-Order SQLis, which are notoriously difficult to spot.

Note that we do not search for new SQLi payloads but rather we exploit attacks related to SQLi. This allows us to analyze how an intruder can violate a security property by exploiting one or more attacks related to a SQLi, e.g., credential bypass. Nevertheless, we also (automatically) test our attacks against the web app under analysis and then we use state-of-the-art tools (i.e., sqlmap and curl) to detect the actual payload of all the SQLis exploited.

Organization. In § 2, we discuss a concrete example that shows why we can't stop at the identification of a SQLi. In § 3, we give a categorization of SQLi vulnerabilities, based on which, in § 4, we provide our formalization. In § 5, we discuss SQLfast and its application real-world case studies. In § 6, we sum up and discuss related and future work. The extended version [10] contains full details on our specifications and case studies, and a proof that the formalization of the database correctly handles all the SQLis categorized in § 3.

2 Why can't we stop at the identification of a SQLi? The case of Joomla!

The identification of a SQLi entry point is generally considered as a satisfactory finish line when dealing with SQLi in web apps. So, one could ask: why not simply stop there (and why bother reading the rest of this paper)? The answer is that a SQLi can be a serious threat *only* if it can be exploited and *only* if it

can be used for carrying out an attack. A full understanding of how a potential SQLi vulnerability can afflict the security of a web app is essential in order to implement proper countermeasures. For instance, consider Joomla! [22], a PHP-based Content Management System that allows users to create web apps through a web interface. Joomla! supports different databases, e.g., MySQL [26] and PostgreSQL [30], and a recent assessment [19] has shown that versions ranging from 3.2 to 3.4.4 suffer from a SQLi vulnerability [7].

The execution of a state-of-the-art scanner such as sqlmap on Joomla! can correctly find the vulnerability. However, sqlmap (or any other scanner for SQLi) cannot tell how that SQLi can be usefully exploited in order to carry out a concrete attack. A general description of the consequences of a SQLi attack is given in [8,28] but, whenever a SQLi entry point is found, the penetration tester has to manually investigate the kind of damages that SQLi might cause to the web app. The researchers who discovered the vulnerability of Joomla! [19] also described how it could be exploited in a real attack: it would allow an intruder to perform a session hijack and thus steal someone's session but would not allow him to create his own account or modify arbitrary data on the database. The exploration of different attack scenarios has been entirely performed manually since no automatic tool shows the outcome of the exploitation of a SQLi vulnerability on a specific web app. But who guarantees that a post-SQLi attack can actually be performed and that all possible attacks based on the SQLi have been taken into account by the penetration tester?

This is why we can't stop at the identification of a SQLi and why we can't address the post-SQLi attacks with a manual analysis. Our approach addresses this by automating the identification of attacks that exploit a SQLi.

3 SQL injections

Some general classifications based on the payloads of the SQLi (and the exploitation scenarios) have been put forth, e.g., [15,27]. Based on these, we can divide SQLi techniques into 6 different categories: (i) Boolean-Based, (ii) Time-Based, (iii) Error-Based, (iv) `UNION` Query, (v) Second-Order and (vi) Stacked Queries.

Given that our formalization strictly depends on the attack that the intruder wants to perform by using a particular type of SQLi, we now define the two attacks that we have considered:[4]

- *Authentication bypass attack*: the intruder bypasses an authentication check that a web app performs by querying a database.
- *Data extraction attack*: the intruder obtains data from the database that he should not be able to obtain.

Based on these attack definitions, we will now describe the main details of each category, emphasizing those aspects that are relevant for our formalization. The following table summarizes which attacks can be exploited by a SQLi technique on a specific type of SQL query. Three remarks: (1) since all state-of-the-art

[4] Other possible attacks (e.g., by exploiting a *Cross-Site Scripting (XSS)* inside the payload of some SQLi) are outside the scope of our approach for now, cf. § 6.

DBMS are vulnerable to SQLi, we won't distinguish between different dialects of SQL and simply write "SQL query"; (2) `AB` abbreviates *authentication bypass* and `DE` *data extraction*; (3) a scenario in which the intruder extracts information in order to bypass an authentication is considered to be a data extraction attack.

	BB		TB		EB		UQ		SO		SQ	
	AB	DE	AB	DE	AB	DE	AB	DE	AB	DE	AB	DE
SELECT	✓	✓	✓	✓	✓	✓	✓	✓				✓
UPDATE	✓	✓	✓	✓	✓	✓			✓	✓		✓
DELETE						✓						✓
INSERT						✓			✓	✓		✓

In a **Boolean-Based SQLi (BB)**, an intruder inserts into an HTTP parameter, which is used by a web app 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 app and comparing the responses, the intruder can understand whether or not the injection was successful. In this way, an intruder can perform both authentication bypass and data extraction attacks.

In an authentication bypass attack, the intruder injects a statement that changes the truth value of a `WHERE` clause in a SQL `SELECT` query, creating a tautology. If a web app performs an authentication check querying a database, this attack will then trick the database into replying in an affirmative way even when no (or wrong) authentication details have been presented by the intruder.

In a data extraction attack, the intruder obtains data from the database. The term "extraction" is used in standard terminology but it can be misleading. With a BB, an intruder exploits the "Boolean behavior" of a web app inferring whether the original query returned some tuples or not. When the intruder understands how the web app behaves when some tuples or no tuples are returned, he can start the "extraction". In this case, the intruder asks whether a certain information is stored in the database and, based on the behavior of the web app, he knows if the information is actually inside the database.

A **Time-Based SQLi (TB)** is quite similar to BB: the only difference is that TB does not need the web app to have a Boolean behavior. The intruder appends a timing function to the validity value of a Boolean clause. Thus, after the submission of the query by the web app, the database waits for a predefined amount of time for a tuple as a response to the query; the intruder can then infer whether the Boolean value of the query was true or false observing a delay in the response. In real case scenarios, a BB is preferable as it is faster than a TB. Timing is not part of our formalization (see § 4), so the abstract attack traces generated by our tool will not distinguish between BB and TB.

When error pages are exposed to the Internet, some error messages of the database could be exposed, thus giving an intruder the possibility of exploiting an **Error-Based SQLi (EB)**. In this type of injection, the intruder tricks the database into performing operations that result in an error and then he extracts information from the error messages produced by the database. EB is generally used to perform a data extraction attack by inducing the generation of an error that contains some information stored in the database.

A `UNION` **Query-Based SQLi (UQ)** is a technique in which an intruder injects a SQL `UNION` operator to join the original query with a malicious one.

The aim is to overwrite the values of the original query and thus, in order to extract information, UQ requires the web app to print the result of the query within the returned HTML page. This behavior allows the intruder to actually extract information from the database by reading it within the web app itself.

Second-Order SQLi (SO) is an injection that has no direct effect when submitted but that is exploited in a second stage of the attack. In some cases, a web app may correctly handle and store a SQL statement whose value depends on the user input. Afterwards, another part of the web app that doesn't implement a control against SQLi might use the previously stored SQL statement to execute a different query and thus expose the web app to a SQLi. Automated scanners generally fail to detect this type of SQLi (e.g., [33,34]) and may need to be manually instructed to check for evidence that an injection has been attempted.

With a **Stacked Queries SQLi (SQ)**, an intruder can execute an arbitrary query different from the original one. The semicolon character ; enables the intruder to concatenate a different SQL query to the original one. By doing so, the intruder can perform data extraction attacks as well as execute whatever operation is allowed by the database. With a SQ, an intruder can perform any of the SQLis described above. Thus, whenever we refer to all the SQLis in our categorization, we exclude SQ as it is already covered by the other ones.

Prevention techniques. Avoiding SQLi attacks is theoretically quite straightforward. In fact, developers can use sanitization functions or prepared statements. Roughly speaking, the general idea is to not evaluate the injected string as a SQL command.

A *sanitization function* takes the input provided by the user and removes (i.e., escapes) all the special characters that could be used to perform a SQLi. Sanitization functions are not the best option when dealing with SQLi because they might not be properly implemented or do not consider some cases.

Prepared statements are the best option for preventing SQLis. They are mainly used to execute the same query repeatedly maintaining efficiency. However, due to their inner execution principle (if properly implemented) they are immune to SQLi attacks. The execution of a prepared statement consists mainly in two steps: preparation and execution. In the preparation step, the query is evaluated and compiled, waiting for the parameters for the instantiation. During the execution step, the parameters are submitted to the prepared statement and handled as data and thus they cannot be interpreted as SQL commands.

4 A Formalization of SQLi

We will now describe how we formally represent a web app that interacts with a database using insecure SQL queries and/or a sanitized (i.e., secure) query. In § 4.1, we propose an extension of the DY model that can deal with SQLi.[5] We formalize the database in § 4.2, the web app in § 4.3, and the goals in § 4.4. For

[5] This formal representation is intended to work with tools that perform symbolic analysis. We don't formalize the honest client behavior and we assume the DY intruder to be the only agent able to communicate with the web app. The DY intruder

brevity and readability, we omit many details and only give pseudo-code that should be quite intuitive. See [10] for full details and the ASLan++ code of our formalizations and case studies, along with a brief introduction to ASLan++.

4.1 The DY web intruder

We extend the standard DY intruder model [11] for security protocol analysis. Suppose that we want to search for an authentication bypass attack via BB (§ 3), in which the intruder injects a statement that changes the truth value of a `WHERE` clause in a SQL `SELECT` query, creating a tautology. To formalize this, we need to extend the DY intruder by giving him the ability to send a concatenation of Boolean formulas made of conjunctions and disjunctions. This characteristic highlights an important difference between the classical DY intruder and the enhanced version we are proposing: *our web intruder works with abstract payloads rather than messages*. Due to technical details (e.g., implementation constraints and non-termination problems), implementing such a modification is impractical. We have thus allowed the intruder 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, it evaluates to true.

We can rephrase the same reasoning in the case of BB for data extraction attacks, in which the intruder tricks the web app into asking to the database if a particular information is present; for example, instead of `or.true`, the intruder adds `or username=admin`. The DBMS will reply in an affirmative way only if there is a tuple in the database with `admin` as username. To allow the intruder to perform all the SQLis described in § 3, we thus extend the DY intruder with one constant `sqli` that represents any SQLi payload (e.g., `or.true`).

4.2 The database

We give a general formalization of a database that can be used in any specification to exploit SQLi when searching for security flaws in a web app. Our formalization aims to be both *compact*, to avoid state-space explosion problems, and *general enough* not to be tailored to a given technology (e.g., MySQL or PostgreSQL). Hence, we don't represent the database content, the database structure, the SQL syntax nor access policies specified by the DBMS. Rather, we formalize messages sent and received and queries, and a database can be seen as a network node that interacts only with the web app through a secure channel.[6]

Definition 1. Messages *consist of* variables V, constants c *(`sqli`, etc.)*, concatenation $M.M$, function application $f(M)$ *of uninterpreted function symbols*

will eventually perform honest interactions if needed to achieve a particular configuration of the system. See [10] for more details.

[6] Nothing prevents us from relaxing this assumption but this would give the DY intruder the possibility of performing attacks (e.g., man-in-the-middle attacks) that are rare in web app scenarios.

f to messages M (e.g., `tuple`(M)*), and* encryption $\{M\}_M$ *of messages with public, private or symmetric keys that are themselves messages. We define that* M_1 *is a* submessage *of* M_2 *as is standard (e.g.,* M_1 *is a submessage of* $M_1.M_3$*, of* $f(M_1)$ *and of* $\{M_1\}_{M_4}$*) and, abusing notation, write* $M_1 \in M_2$*.*

Definition 2. *A query is* valid *(respectively,* not valid*) when, evaluated by a database, it returns one or more (respectively, zero) tuples.*

We formalize the validity of SQL queries by means of the Horn clause: `inDB(M.sqli)` $\implies$ *true*, where, in order to represent a SQLi attack, the predicate `inDB()` holds for a message (which represents a SQL query) whenever it is of the form M.`sqli`. This states that the intruder has injected a payload `sqli` into the query parameters (expressed as a variable) M.

Incoming messages. We consider, as incoming messages, only SQL queries via raw SQL and via sanitized queries. The parameters of queries are represented by a generic variable `SQLquery`. In case of a raw SQL query, they are wrapped by an uninterpreted function `query()`; if a sanitized query has been implemented then we use another uninterpreted function `sanitizedQuery()`. These two uninterpreted functions allow the modeler to "switch on/off" the possibility of a SQLi in some point of the app.

Database responses. The tuple generated by the database as a response to a raw SQL query is represented by an uninterpreted function `tuple()` over a message representing a SQL query. Given that we do not model the content of the database, this function represents any (and all) database data.

Whenever the database receives a SQL query `query(SQLquery)` from the web app, the uninterpreted function `tuple(SQLquery)` is sent back to the web app to express that a tuple, as a response to the query, has been found. This response is returned only if `inDB()` holds; in all other cases, a constant `no_tuple` is returned to represent that no tuples are returned in the responses of the database.

If the database receives a sanitized query, no injection is possible. Hence, the database does not return any useful information to the web app; instead, a constant `no_tuple` is returned. Since the intruder cannot perform a SQLi in presence of a sanitized query, we also assume that a sanitized query can be executed only with legitimate parameters, i.e., as a function of `tuple()` (this is because we are interested in modeling only SQLi scenarios).

The pseudo-code representing the database behavior is given in Listing 1.1, where, here and in the following, we write `DB` for the database. `DB` is a network node and we assume it to be always actively listening for incoming messages. It is defined by two main, mutually exclusive, branches of an if-elseif statement: one guard is in line 1 in which `DB` is waiting (expressed in Alice-and-Bob notation) for a sanitized query and the other in line 3 in which it is waiting for a raw SQL query. If a sanitized query is received, then there is no SQLi. Given that we only consider dishonest interactions, the data sent back to the intruder will not increase his knowledge. In other words, no SQLis are permitted and any permitted query will just give to the intruder the possibility of continuing his execution with the web app but won't add any extra information to his knowledge.

Listing 1.1. Pseudo-code of a DBMS.

```
1  if(WebApp -> DB: sanitizedQuery(SQLquery)){
2   if(SQLquery == tuple(*)) DB -> WebApp: no_tuple;
3  }elseif(WebApp -> DB: query(SQLquery)){
4   if(inDB(SQLquery)) DB -> WebApp: tuple(SQLquery);
5   if(!(inDB(SQLquery))) DB -> WebApp: no_tuple;}
```

One may argue that a valid query should indeed add extra information to the intruder knowledge. However, we do not model the content of the database and any information received by the intruder as a response to a sanitized query is included in the action that the web app performs after this database response. Thus, in our formalization, the query in `SQLquery` is not valid and then the `no_tuple` constant is sent back in line 2. We also add a constraint in line 2 that any query received (`SQLquery`) must be of the form `tuple(*)`, i.e., as a function of the content of the database where `*` acts as a wildcard character that matches any possible parameter. This is because, in the case of a sanitized query, the intruder cannot perform a SQLi and we exclude the case in which the DY intruder sends a random query just to continue the execution with the web app. Instead, he has to either know a tuple of the database or data as functions of a tuple of the database. In the case the intruder knows `tuple(Query)`, he will just receive `no_tuple`, i.e., correctly no data has been leaked to the intruder.

The second branch of the initial if-elseif statement (line 3) handles raw queries. If a raw query is submitted, then there are two cases: the raw query is not valid (line 5, where `!` formalizes the negation) and then, as in the previous case, `no_tuple` is sent back (line 5); the raw query is valid (line 4) and a tuple is sent back (line 4). Given that all these queries are sent from the intruder, we can assume they have a malicious intent. One may argue that, in a real case scenario, the database is not actually returning a tuple but, given that an intruder could repeatedly send a SQLi exploiting that injection point, it is fair to assume that the database is sending all the tuples it contains, i.e., `tuple(SQLquery)`.

4.3 The web app

As for the database, the web app is a node of the network that can send and receive messages. The web app communicates with a client or with the database (it can, potentially, also communicate with other apps but we do not consider that explicitly here). We assume only one database is present because adding other databases would not add any further useful information for finding attacks based on SQLi. The proof is straightforward. Since we do not consider database contents and structures, if we wanted to have two database models, then we would have two exact copies of the formalization given in § 4.2. Since we have assumed that there exists a long-lasting secure relation between the database and the web app, no man-in-the-middle attacks are considered. Therefore, any attack found that involves the communication with one of the two databases could be found by considering the other database only.

A specification of a web app can be seen as a behavioral description of the web app itself (along with its interaction with the database). A modeler can define this specification from the design phase documentation of the engineering

process of the web app. A model can also be created in a black box way by just looking at the HTTP messages exchanged from a client and the web app and guessing the communication with the database.

We now consider the main aspects that allow for the modeling of a web app.

Sending and receiving messages. A web app can communicate with a client and the database. We abstract away as many details as possible of the web pages and thus any incoming message will only contain: (i) parameters of forms expressed as variables, e.g., `Client -> WebApp: Username.Password`, and (ii) the web page itself expressed as a constant, e.g., `WebApp -> Client: dashboard` where `dashboard` represents a web page. Note that, in any response of the web app, if the content of the response is linked to a response of the database, i.e., `tuple(Query)` (where the query is either `SELECT`, `UPDATE` or `DELETE`), then `tuple(Query)` must be included in the response. Otherwise, we would end up representing a scenario in which no content of the tuple received by the database is included in, or linked to, the web page and thus no SQLi would be present.

Queries. The web app creates either a sanitized query or a raw query. Then, the web app wraps the variables representing the query parameters with either `sanitizedQuery()` or `query()`, both uninterpreted functions, and afterwards sends the SQL query to the database. Note that we only need to represent the parameters of a SQL query since we do not distinguish between different queries in the database formalization. If the query does not depend on parameters sent from a client, the intruder cannot exploit it to perform a SQLi. The SQL query used to query the database is represented as a constant, resulting in the database always replying with `no_tuple` (as `inDB()`, in this case, is never valid).

If statements. We use them mainly to decide, based on which kind of message has been received, what the web app has to reply. For example, if the database replies with a tuple `tuple(Query)`, then the web app might return a specific page along with `tuple(Query)` or might return a different page.

Assignments. A constant or a message can be assigned to a variable: `Variable := constant|message`. Assignments are, e.g., useful to save incoming messages.

4.4 Goals

Finally, we define the security properties we want the model to satisfy. As we discussed in § 3, we consider two main attacks: authentication bypass and data extraction. We give the formalization in Listing 1.2, where `iknowledge` is a predicate that represents the knowledge of the intruder. By using the LTL "globally" operator `[]`, we can specify an authentication check by stating that the intruder should not have access to a specific page (`dashboard` in Listing 1.2), whereas data extraction is represented by specifying that the intruder should not increase his knowledge with data from the database (i.e., as function of `tuple()`).

Listing 1.2. Authentication bypass and data extraction goals of the BB example.

```
[](!(iknowledge(dashboard))); %authentication bypass
[](!(iknowledge(tuple(*))));  %data extraction
```

5 SQLfast, Case Studies and Results

To show that our formalization can be used effectively to detect security flaws linked to SQLi attacks, we have developed *SQLfast*, a prototype *SQL Formal AnalysiS Tool* [32]. In [32] we also provide a friendly web-based user interface that helps the modeler in creating the web-app model. SQLfast takes in input a specification written in ASLan++, the modeling language of the AVANTSSAR Platform [3], and then calls CL-AtSe (one of the platform's model checkers) and generates an *Abstract Attack Trace (AAT)* as a *Message Sequence Chart (MSC)* if an attack was found. SQLfast automatically detects which type of SQLi was exploited and, in an interactive way, generates the curl or sqlmap commands to concretize the attack.

As a concrete proof-of-concept, we have applied SQLfast to (i) WebGoat [29], (ii) Damn Vulnerable Web Application (DVWA) [13], (iii) Joomla! 3.4.4, and (iv) *Yet Another Vulnerable Web Application (YAVWA)*, an ad-hoc testing environment that we have developed and that also includes a SO SQLi example. (Recall that full details are given in [10].) The case studies provided by WebGoat and DVWA might sound limited but capture all possible scenarios with respect to SQLi attack combinations considered in this paper — recall that our formalization for SQLi attacks does not find SQLi payloads, but focuses on vulnerabilities based on SQLi. We tested SQLfast in order to show all the combinations that could be represented by considering SQLi for (1) authentication bypass, (2) data extraction and (3) data extraction with reuse of the extracted information. Our case studies are quite heterogenous, so it should not be difficult to map other case studies to one of these scenarios we have considered.

We have implemented the case studies in ASLan++ to be able to apply the model checkers of the AVANTSSAR Platform (in particular, CL-AtSe), but other model checkers implementing the Dolev-Yao intruder model could be used as well, provided that their input language is expressive enough. For the sake of brevity, we discuss here only the case studies Joomla!, YAVWA and SO, which show how our formalization can find attacks linked to the logic of a web app that is vulnerable to SQLi attacks. The type of attacks that SQLfast can detect and concretize cannot be detected by state-of-the-art tools for SQLi such as sqlmap.

5.1 Case Study: Authentication Bypass via Data Extraction

We now discuss two scenarios in which our approach detects attacks that state-of-the-art tools, such as sqlmap, are *not* able to detect and exploit. In the first scenario, the intruder exploits a recent SQLi vulnerability found (by manual inspection only) in Joomla! [7]. The second scenario (YAVWA) is a variant of the first and shows a concatenation of different attacks.

Joomla!. A recent assessment has shown that the Content History module of Joomla! suffers from a SQLi vulnerability that allows a remote (non-authenticated) user to execute arbitrary SQL commands [7]. The pseudo-code in Listing 1.3 represents the following behavior: a remote user visits the Content

History component (line 1). The web app queries the database with the user supplied data (2). If some tuples are generated (3), the web app sends to the client the history page `viewHistory` along with the `tuple()` function (4). The web app then has two possible ways of authenticating the user (5–9): by using credentials or cookies. If username and password are provided (5), the web app applies a non-invertible hash function `hash()` to the password, and queries the database to verify the credentials (6).[7] If the credentials are correct, the administration panel is sent to the user (7). In case of a cookie session, the user provides a cookie that the web app checks querying the database (8). If the cookie is valid, the administration panel is sent back to the user (9).

Listing 1.3. Pseudo-code representing the Joomla! scenario.

```
1  User -> WebApp: com_contenthistory.history.Listselect;
2  WebApp->DB: query(com_contenthistory.history.Listselect);
3  if(DB -> WebApp: tuple(SQLquery)){
4   WebApp -> User: viewHistory.tuple(SQLquery); }
5  if(User -> WebApp: Username.Password){
6   WebApp -> DB: sanitizedQuery(Username.hash(Password));
7   if(DB -> WebApp: no_tuple){ WebApp -> User: adminPanel; }}
8  if(User -> WebApp: Cookie){ WebApp -> DB: sanitizedQuery(Cookie);
9   if(DB -> WebApp: no_tuple){ WebApp -> User: adminPanel; }}
```

As goal, we check if there exists an execution in which the intruder can access the administration panel represented by the constant `adminPanel`.

Listing 1.4. Authentication bypass for the Joomla! scenario.

```
[](!(iknowledge(adminPanel)));
```

SQLfast generates the AAT in Listing 1.5, which is an authentication bypass attack where the intruder hijacks a user session by using a cookie instead of login credentials. In fact, the web app applies a hash function to the password before verifying the credentials submitted by the user. The hash function would not allow an intruder to blindly submit a password extracted from the database, the only possibility is using a valid cookie value.[8] The intruder performs a data extraction and retrieves the information to access the administration panel (1–4), and uses it to hijack a user session by submitting a valid cookie value (5–8).

Listing 1.5. Abstract attack trace that extracts data with a SQLi in order to bypass the authentication of the Joomla! scenario.

```
1  i -> WebApp  : com_contenthistory.history.sqli
2  WebApp -> DB : query(com_contenthistory.history.sqli)
3  DB -> WebApp : tuple(com_contenthistory.history.sqli)
4  WebApp  -> i : viewHistory.tuple(com_contenthistory.history.sqli)
5  i ->WebApp   : cookie.tuple(com_contenthistory.history.sqli)
6  WebApp -> DB : sanitizedQuery(tuple(com_contenthistory.history.sqli))
7  DB -> WebApp : no_tuple
8  WebApp  -> i : adminPanel
```

YAVWA. We have designed a variant of Joomla! to show that a SQLi can be exploited to compromise a part of a web app that does not directly depend on

[7] The web app applies a hash function to the password before checking whether credentials are correct because Joomla! stores the passwords hashed into the database.

[8] We do not consider the possibility of brute forcing the hashed password, in accordance with the perfect cryptography assumption of the DY model.

databases. YAVWA provides an HTTP form login and a login by HTTP basic authentication [18] configured with the `.htaccess` [2] file. The credentials used for the HTTP basic authentication, which are stored in the `.htpasswd` file, are the same as the ones employed by the users to login into the web app (i.e., the same as the ones stored in the database). The intruder's goal is to access the area protected by the HTTP basic authentication login. Obviously, he cannot perform a SQLi to bypass HTTP basic authentication since the login procedure doesn't use SQL. Bypassing the login page, without knowing the correct credentials, doesn't allow the intruder to gain access to the secure folder.

We have defined this scenario in the pseudo-code in Listing 1.6. The client sends his personal credentials (`Username.Password`) to the web app (1). The web app creates a query that it sends to the database (2) for verifying the submitted credentials. If tuples are generated from the database (3), a dashboard page is returned to the client along with the function `tuple()` (4), otherwise, the web app redirects the user to the login page (5). At this point, the web app waits to receive correct credentials that will allow the client to access the secure folder `secureFolder` (6). Given that the credentials are the same as the ones stored in the database, and the database content is represented with the function `tuple()`, we can also represent credentials here with the function `tuple()`.[9]

Listing 1.6. Pseudo-code representing the YAVWA scenario.

```
1  User -> WebApp: Username.Password;
2  WebApp -> DB: query(Username.Password);
3  if(DB -> WebApp: tuple(SQLquery)){
4   WebApp -> User: dashboard.tuple(SQLquery);
5  }elseif(DB -> WebApp: no_tuple){ WebApp -> User: login; }
6  if(User -> WebApp: tuple(*)){ WebApp -> User: secureFolder; }
```

As goal, we check if the intruder can reach `secureFolder`. SQLfast generates the AAT given in Listing 1.7, in which the intruder successfully retrieves information from the database and uses such information to access a protected folder. The intruder performs a data extraction attack using SQLi (1–4), which allows him to retrieve information stored in the database, and then (5–6) submits the extracted data and accesses the restricted folder `secureFolder`.

Listing 1.7. Abstract attack trace of the YAVWA case study.

```
1  User -> WebApp: Username(4).sqli
2  WebApp -> DB  : query(Username(4).sqli)
3  DB -> WebApp  : tuple(Username(4).sqli)
4  WebApp -> i   : dashboard.tuple(Username(4).sqli)
5  i  -> WebApp  : tuple(Username(4).sqli)
6  WebApp  -> i  : secureFolder
```

5.2 Case Study: Second-Order SQLi (SO)

We now show that our formalization is flexible enough to represent SOs, which are notoriously very difficult to detect and exploit.

[9] We recall from § 4.2 that `tuple()` represents an abstraction of any data that can be extracted from the database. This means that whenever a web app requires any data in the domain of the database, we can write them as a function of `tuple()`.

This scenario is part of YAVWA and implements a web app that allows users to register a new account. In the registration process, the web app executes an (`INSERT`) SQL query that stores the user's credentials into a database. The intruder can create an account submitting malicious credentials that don't result in a SQLi but will trigger an injection later on in the web app. After the registration phase, the user submits a request for accessing an internal page. The web app performs another SQL query using the same parameters previously used in the registration process (i.e., the registration credentials). At this point, a page is showed together with the injection and the intruder can exploit a SO.

We have formalized this scenario in Listing 1.8: a client sends a registration request along with his personal credentials (`Username` and `Password`) to the web app (1). The web app sends a query containing the client's credentials to the database (2). The web app checks if it receives a response from the database containing the data resulting from the execution of the query `tuple(SQLquery)` submitted by the web app (3). The web app sends back to the client the page `registered` (4). Here, the web app does not forward `tuple()` because the registration query is an `INSERT` (see § 4.2). The client asks for a page (5), which makes the web app use previously submitted values of `Username` and `Password` to execute a new SQL query (6). Here is where the SO takes place; the variables embedded in the query in (6) will trigger a SO. The database executes the query and sends back the results to the web app (7). Finally (8), the web app sends (by using a `SELECT` query) to the client the requested page and the `tuple()`.[10]

Listing 1.8. Pseudo-code representing a web app vulnerable to a SO attack.

```
1  User -> WebApp: registrationRequest.Username.Password;
2  WebApp -> DB: query(Username.Password);
3  if(DB -> WebApp: tuple(SQLquery)){
4   WebApp -> User: registered;
5   User -> WebApp: requestPage;
6   WebApp -> DB: query(Username.Password);
7   DB -> WebApp: tuple(SQLquery);
8   WebApp -> User: page.tuple(SQLquery); }
```

As goal, we ask if the intruder can interact with the web app until he obtains data from the database, i.e., with a data extraction attack, as in Listing 1.2. SQLfast generates the AAT in Listing 1.9, in which the intruder performs the registration process (1–4) by registering malicious credentials `Username(4)` and `sqli`. At the end of the registration process (5), the intruder asks for `requestPage` that makes the web app send to the database a SQL query with the same parameters the intruder used in the registration (6–7). In (8), the intruder receives the requested page and the result of the execution of the injected SQL query performing a SO.

Listing 1.9. Abstract attack trace for the SO case study.

```
1  User -> WebApp: registrationRequest.Username(4).sqli
2  WebApp -> DB  : query(Username(4).sqli)
3  DB -> WebApp  : tuple(Username(4).sqli)
4  WebApp -> i   : registered
5  i -> WebApp   : requestPage
```

[10] Recall that we don't represent SQL syntax in our models, so we don't explicitly represent the type of the SQL according to the modeling guidelines in § 4.3.

```
6  WebApp -> DB  : query(Username(4).sqli)
7  DB -> WebApp  : tuple(Username(4).sqli)
8  WebApp -> i   : page.tuple(Username(4).sqli)
```

5.3 Concretization phase

We executed SQLfast on all our case studies using a standard laptop (Intel i7 with 8G RAM). The execution time of the model-checking phase of SQLfast ranges from 35 to 45 ms. The overall process (from translation to concretization) takes a few seconds. In all the cases, we generated AATs violating the security property we defined over the model (authentication bypass or data extraction attack). Once the AAT has been generated, SQLfast interactively asks the user to provide information such as the URL of the web app. Finally, if we are concretizing a SQLi that exploits an authentication bypass attack a curl command is showed, whereas sqlmap is used for data extraction SQLi. By executing the traces generated by SQLfast, we exploited all the AATs over the real web app.

6 Conclusions, related work, and future work

We have presented a formal approach for the representation of SQLi and attacks that exploit SQLi in order to violate security properties of web apps. We have formally defined web apps that interact with a database (that properly replies to queries containing SQLi) and an extended DY intruder able to deal with authentication bypass and data extraction attacks related to SQLi. We have shown the efficiency of our prototype tool SQLfast on four real-world case studies (see also [10]). SQLfast handles SO and detects multi-stage attacks and logical flaws that, to the best of our knowledge, no other tool can handle together, and hardly ever even individually, including the discovery of an attack on Joomla!.

Many works have proposed new SQLi techniques and payloads (e.g., [35,21,9]) or formal approaches to detect SQLi (e.g., [25,23,16,24]). However, to the best of our knowledge, ours is the first attempt to search for vulnerabilities based on SQLi rather than to detect SQLi. There are, however, a number of works that are closely related to ours and that are thus worth discussing.

SPaCiTE is a model-based security testing tool for web apps that relies on mutation testing [4]. SPaCiTE starts from a secure ASLan++ specification of a web app and automatically introduces flaws by mutating the specification. The strength of this approach is the concretization phase. Starting from an AAT, generated from the mutated specification using a model-checking phase, SPaCiTE concretizes and tests the attack trace on the real web app. The major differences with respect to our approach reside in how we model web apps and in particular those aspects that strictly characterize SQLi aspects. The main goal of the approach in [4] is to find SQLi entry points and concretize them, our main goal is to consider SQLi aspects that can be exploited to attack a web app.

Another formal approach that uses ASLan++ and the DY intruder model for the security analysis of web apps is [31]. In this work, the authors model a web app searching for CSRF and they do not consider databases or extensions to the

DY model. However, the idea and the representation of web apps is close to ours and we envision some potentially useful interaction between the two approaches.

In [5], the authors describe the "Chained Attack" approach, which considers multiple attacks to compromise a web app. The idea is close to ours, but: (i) they consider a new kind of web intruder, whereas we stick with the DY intruder; (ii) we analyzed the most common SQLi techniques and proposed a formalization of a vulnerable database, they only consider the behavior of the web app.

In [1], the authors present a model-based method for the security verification of web apps. They propose a methodology for modeling web apps and model 5 case studies in Alloy [20]. Even if the idea is similar to our approach, they have defined three different intruder models that should find web attacks, whereas we have used (and extended) the standard DY one. Their AATs are difficult to interpret because no MSCs are given but state configurations. They have also considered a number of HTTP details that we have instead abstracted away in favor of an easier modeling phase. In contrast, we display AAT as MSCs and we proposed a concretization phase to obtain the concrete payloads of SQLi.

As future work, we plan to extend the database formalization in order to consider SQLi that would modify the database state leading to more complex SQLi exploitations. We also plan to analyze other web app vulnerabilities such as stored/reflected XSS and broken session management, and investigate synergies between our approach and the one of [31] on CSRF. We will extend our approach to detect (i) complex concatenations of vulnerabilities (similar to, and more complex than, [17]) that lead to concatenations of attacks, and (ii) articulated paths to vulnerabilities that would hardly ever be discovered by manual analysis.

References

1. D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF*, pages 290–304. IEEE, 2010.
2. Apache software foundation. Apache HTTP Server Tutorial: .htaccess files. `https://httpd.apache.org/docs/current/howto/htaccess.html`.
3. A. Armando et al. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS 2012*, LNCS 7214, pages 267–282. Springer, 2012.
4. M. Büchler, J. Oudinet, and A. Pretschner. Semi-automatic security testing of web applications from a secure model. In *SERE*, pages 253–262, 2012.
5. A. Calvi and L. Viganò. An Automated Approach for Testing the Security of Web Applications Against Chained Attacks. In *ACM/SIGAPP SAC*. ACM Press, 2016.
6. S. Christey. The 2009 CWE/SANS Top 25 Most Dangerous Programming Errors. `http://cwe.mitre.org/top25`.
7. CVE-2015-7857. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7857`.
8. CWE. CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). `https://cwe.mitre.org/data/definitions/89.html`.
9. B. Damele and A. Guimarães. Advanced SQL injection to operating system full control. In *BlackHat EU*, 2009.

10. F. De Meo, M. Rocchetto, and L. Viganò. Formal Analysis of Vulnerabilities of Web Applications Based on SQL Injection (Extended Version). http://sqlfast.altervista.com/formal-analysis-web-apps-extended.pdf, 2016.
11. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–208, 1983.
12. A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *DIMVA*, LNCS 6201, pages 111–131. Springer, 2010.
13. Damn Vulnerable Web Application (DVWA). http://www.dvwa.co.uk.
14. J. Forristal. ODBC and MS SQL server 6.5. *Phrack*, 8(54), 1998.
15. W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *SIGSOFT '06/FSE-14*, 2006.
16. W. G. J. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *ASE*, pages 174–183. IEEE, 2005.
17. E. Homakov. How I hacked Github again, 2014. http://homakov.blogspot.it/2014/02/how-i-hacked-github-again.html.
18. Internet Engineering Task Force (IETF). HTTP Authentication: Basic and Digest Access Authentication, 1999. https://www.ietf.org/rfc/rfc2617.txt.
19. iSpiderLabs. Joomla SQL Injection Vulnerability Exploit Results in Full Administrative Access, 2015. https://www.trustwave.com/Resources/SpiderLabs-Blog/Joomla-SQL-Injection-Vulnerability-Exploit-Results-in-Full-Administrative-Access/?page=1&year=0&month=0.
20. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Pr., 2012.
21. O. M. Jayathissa. SQL Injection in Insert, Update and Delete Statements.
22. Joomla! https://www.joomla.org.
23. A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, pages 199–209. IEEE, 2009.
24. V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX*, pages 18–18, 2005.
25. M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *USENIX*, pages 31–43, 2008.
26. MySQL. https://www.mysql.com.
27. OWASP. Owasp top 10 for 2013. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.
28. OWASP. SQL Injection. https://www.owasp.org/index.php/SQL_Injection.
29. OWASP. WebGoat Project. https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project.
30. PostgreSQL. http://www.postgresql.org.
31. M. Rocchetto, M. Ochoa, and M. Torabi Dashti. Model-Based Detection of CSRF. In *IFIP SEC*, pages 30–43. Springer, 2014.
32. SQLfast: SQL Formal AnalisyS Tool, 2015. http://sqlfast.altervista.com.
33. sqlmap: Automatic SQL injection and database takeover tool, 2013. http://sqlmap.org.
34. sqlninja: a SQL Server injection & takeover tool, 2013. http://sqlninja.sourceforge.net.
35. M. Stampar. Data Retrieval over DNS in SQL Injection Attacks. http://arxiv.org/abs/1303.3047, 2013.
36. L. Viganò. The SPaCIoS Project: Secure Provision and Consumption in the Internet of Services. In *ICST*, pages 497–498, 2013.