A Learning-Based Approach to Secure Web Services from SQL/XPath Injection Attacks

Nuno Laranjeiro, Marco Vieira, Henrique Madeira
CISUC, Department of Informatics Engineering
University of Coimbra
Portugal
{cnl, mvieira, henrique}@dei.uc.pt

Abstract—Business critical applications are increasingly being deployed as web services that access database systems, and must provide secure operations to its clients. Although the open web environment emphasizes the need for security, several studies show that web services are still being deployed with command injection vulnerabilities. This paper proposes a learning-based approach to secure web services against SQL and XPath Injection attacks. Our approach is able to transparently learn valid request patterns (learning phase) and then detect and abort potentially harmful requests (protection phase). When it is not possible to have a complete learning phase, a set of heuristics can be used to accept/discard doubtful cases. Our mechanism was applied to secure TPC-App services and open source services. It showed to be extremely effective in stopping all tested attacks, while introducing a negligible performance impact.

Keywords—Web services; security; SQL/XPath Injection; vulnerabilities; code instrumentation.

I. INTRODUCTION

The security of web applications is, in general, quite poor [19]. To prevent vulnerabilities, developers should use best coding practices, perform security reviews of the code, execute penetration tests, use code vulnerability analyzers, etc. However, many times, developers focus on implementing functionalities and on satisfying user requirements (and time-to-market constraints), and disregard security aspects.

A recent McKinsey report points out web services and SOA as key trends in modern software development [13]. Web services are self-describing components that can be used by other software across the web in a platform-independent manner, and are supported by standard protocols such as SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery, and Integration) [5]. However, web services are so widely exposed that any security vulnerability will most probably be uncovered and exploited by hackers.

Command injection attacks are very frequent in the web environment [19]. These attacks take advantage of improperly coded applications to inject and execute commands specified by the attacker in the vulnerable application, enabling, for instance, access to critical data. Vulnerabilities allowing SQL Injection and XPath Injection are particularly relevant in web services [12], as these frequently use a data persistence solution [23]. Besides traditional relational databases, major database vendors and several open-source efforts currently provide XML databases (e.g., Oracle XML DB, SQL Server 2008, Apache Xindice) that typically use XPath expressions to access data. While the goal of XPath Injection is to maliciously explore vulnerabilities in XPath expressions (used, for instance, to access an XML database), SQL Injection tries to change SQL statements in a similar manner [4].

Although web services are increasingly being used in business-critical systems, current development tools do not provide practical ways to protect applications against command injection attacks. We have recently discussed, in a short paper [14], the need for protecting web services against SQL/XPath Injection attacks and proposed a very preliminary approach in that direction. In the present paper we propose a phased approach that is able to learn the profile of regular SQL/XPath requests by transforming those into invariant statements and protect web service applications from SQL/XPath injection attacks by: 1) matching incoming requests with the valid set of requests gathered in the learning phase; 2) applying a set of heuristics when unlearned data access statements appear (i.e., no match can be made with previously learned statements).

The proposed approach is quite effective, has an extremely low overhead, and does not require any access to source code as it uses bytecode instrumentation (note that this work focuses on source code vulnerabilities and not on any specific security mechanisms, such as authentication or data encryption). This integrated tool is extremely important in two scenarios:

- To help web service developers improving their code. During build, developers can use the tool to automatically inject bytecode to protect their services, reducing coding and testing effort. This is particularly useful for junior developers that frequently focus on the functionally and disregard code security.
- To help system administrators improving the security of already deployed services, as the technique can be easily used to improve existing
The proposed approach was used to secure an implementation of the web services specified by the standard TPC-App performance benchmark [23] and also four services adapted from code publicly available on the Internet (a total of 24 service operations). A large number of security problems have been disclosed and corrected, showing the effectiveness of our approach.

The structure of the paper is as follows. Next section presents some background and related work. Section III presents the technique for fixing security problems and Section IV presents the experimental evaluation. Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

Several efforts have been undertaken for the removal of security vulnerabilities in computer software. A common way to remove SQL/XPath Injection vulnerabilities is to change the vulnerable code and separate the query structure from the input data by using parameterized queries [21]. Such queries are available for typical databases under the form of prepared statements (an SQL structure with placeholders for variables), but also for XML databases (or simply applications that use XPath) under the form of XPath parameterized expressions [4]. However, developers frequently disregard this type of programming constructs, leaving their applications vulnerable to attacks [12].

In [21] an algorithm for removing SQL injection vulnerabilities is proposed. The approach consists of replacing SQL statements by secure prepared statements. Code inspection and static analysis were used to disclose code prone to SQL injection, which was then replaced by automatically generated secure code. However, besides requiring access to source code, several aspects related with non-explicit setting, non-string, or iterator-based SQL structures remain unsolved. Our approach is able to overcome these limitations, as it does not make assumptions about the structures used to build SQL statements.

An automated approach that tries to convert text SQL statements into prepared statements is presented in [20]. The strategy is to remove SQL vulnerabilities by replacing vulnerable code with generated secure code. The presented prototype was able to remove SQL injection vulnerabilities in five different statement configurations contained in five custom-built toy projects. However, the conversion algorithms are limited as there is a large number of cases that cannot be handled (e.g., handling batch SQL statements).

A proposal for attack injection prevention is presented in [11]. In this case, vulnerabilities are avoided by construction. The proposed methodology consists of embedding the syntax of the guest languages into the syntax of the host language (e.g., SQL in Java) and automatically generating code that maps the embedded language to constructs in the host language that reconstruct the embedded sentences, adding escaping functions where appropriate. Although the approach is generic enough to be adapted to various languages, it obviously adds complexity to the development phase.

AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks) [24] is a tool that uses a model-based approach designed to detect SQL injection attacks, and combines static analysis and runtime monitoring. Static analysis is used to analyze the source code of a given web application, automatically building a model of the legitimate queries that such application can generate. At runtime, AMNESIA monitors all dynamically generated queries and checks them for compliance with the statically generated model. Unlike AMNESIA, we propose learning the profile of legitimate queries at runtime, which may represent a richer, more realistic profile learning, overcoming the intrinsic limitations of static analysis (e.g., requiring access to source code).

An anomaly-based system that learns the profile of normal database accesses in web-based applications is presented in [6]. This system uses a composition of several models that allow the detection of unknown attacks with reduced false positives. However, this approach is currently unable to display information about the coverage space of the training data, which impacts its efficiency. The approach that we propose allowed us to build a high performance mechanism that, in our tests, showed to introduce a negligible performance overhead in typical-size web services.

III. SECURITY IMPROVEMENT APPROACH

To perform SQL Injection the attacker exploits an unchecked input in order to modify the structure of a SQL command [4]. Usually, the attacker starts by adding an extra condition in the ‘where’ clause of a SQL command to gain some form of privileged access. Then the attacker executes a SQL command returning valuable information (typically using a union clause with the malicious select), disrupting the database by performing inserts, deletes or updates. The same happens for XPath Injection (only the syntax differs).

Our proposal to identify potential SQL and XPath injection attacks is based on detecting anomalies, which consists in searching for deviations from an historical (learned) profile of good commands, and includes two major phases:

1. **Statement learning** – Consists of using a real or generated workload (i.e., a set of calls) to exercise the web service. The service is instrumented to learn valid SQL statements and XPath expressions executed during the execution of that workload.

2. **Service protection** – Consists of instrumenting the service to provide protection against SQL/XPath Injection attacks.

A. **Workload Generation**

The first step for the generation of a workload is the **inspection of the service description document**, the WSDL file. This XML file is automatically processed to obtain the list of operations, parameters and associated data types. The information describing the structure and type of all inputs
and outputs of each operation is usually found in a XML Schema file (a XSD file that describes the structure of an XML object), which is referenced by the original WSDL [5]. Additionally, the workload generation tool needs to gather information on the valid domains for all input and output objects. For this, the XSD file, that describes all parameters, is searched. This file may also include information on valid values for each parameter, provided that XSD schema restrictions are defined. It is rare, however, to find the valid values for each parameter expressed in a WSDL/XSD pair. This is due to:

- Lack of integrated tools (and programming language support) that could be easily used to add the domain values to the service’s WSDL descriptor.
- Currently WSDL or XSD have no support for expressing dependencies between multiple parameters of a given service operation. This absent feature impairs the full definition of a domain.

To tackle this last issue, we have recently proposed a language that enables a full domain expression in the XSD file associated with the WSDL [16]. The proposed language, named ‘Extended Domain Expression Language – EDEL’, enables services to fully express their operations domains (including complex parameter domains interdependencies). This can be used to easily create workloads that respect the operations’ domains, hence greatly increasing their coverage.

After having collected the necessary service information, the workload generation is conducted so that we can exercise as many source code points as possible (ideally, the complete set of data access SQL/XPath statements present in the code). In the present work, a set of well-known tools were combined and integrated in an automated synthetic workload generation process depicted in Figure 1.

Some of the tools presented in Figure 1 are specific to Java, but similar ones exist for all major languages. Although the workload generation process has been initially detailed in [16], we include a small description here, as it is part of the protection scheme and therefore important for a full understanding of the approach.

Using the XSD file as starting point, we generate a synthetic workload by using appropriate tools (stage 1), such as benerator (http://databene.org/benerator). This tool is able to read XSD Schema files and, using the domain information present in each schema, can generate a set of XML files containing values used later to exercise our target service. To use the generated values, we need to create programming language level objects that accurately represent the structures found in the XSD file (stage 2). JAXB’s binding compiler (xjc) can be used for this purpose.

At this point, we are able to use XStream (http://xstream.codehaus.org/) to automatically deserialize the produced XML into the corresponding generated Java objects, creating this way a list of objects that form our final workload (stage 3). This is a process that uses reflection to load classes by name and builds a list of objects that are integrated into one unit test case per each service operation (stage 4).

Most tools like benerator are, up to this date, unable to consider multiple domain relations for the input parameters. In fact, to generate the input values this tool only allows the definition of a single domain restriction. Although this restriction can also be a union of restrictions, inter-parameter restrictions are not taken into account, hence not usable. This way, the workload may include invalid service calls that have to be identified and discarded.

A difficulty related to the workload generation is that the coverage of the service calls is not easy to guarantee. This way, the next step consists of executing the workload and using a test coverage analysis tool (e.g. Cobertura [3]) to get a metric of the code coverage (stage 5). If the developer is not satisfied with the coverage then more web service calls are required. Calls must be added to the workload until the coverage reaches the level the developer desires (stage 6).

**B. SQL/XPath Learning**

The first step is to exercise the web service by running the workload and automatically identify all the locations in the web service code where SQL and XPath commands are executed. This is achieved by using AOP (Aspect Oriented Programming) [7] to transparently intercept all calls to a set of method signatures that correspond to well-known APIs for executing SQL commands (e.g., Java’s JDBC API, the Spring Framework JDBC API, etc.) and evaluating XPath expressions (e.g., Java’s JAXP API). This set of APIs is easily extensible; the only requirement is to know the full signature of the method to be intercepted. Figure 2 represents the basic architecture for our interception mechanism. The learning module is described in the following paragraphs, whereas the protection module (see Figure 2) is described in Section III.C.
At runtime, each data access call is intercepted and delivered to a dispatcher. The decision here is simply to check if the application is in learning mode or in protection mode, in each case the request is delivered to an appropriate module (learner or protector module, respectively). During learning, SQL and XPath commands are parsed in order to remove the data variant part (if any). In other words, the information used does not represent the exact command text, since commands may differ slightly in different executions, while keeping the same structure. For example, in the SQL command “SELECT * from EMP where job like ‘CLERK’ and SAL > 1000”, the job and salary in the select criteria (job like ? and sal > ?) depend on the user’s choices. This way, instead of considering the full command text, we just represent the invariant part of it.

Each invariant command is associated with a source code entry point (provided by the AOP framework) in a Map structure. This does not mean that we need the original application’s source code, but it rather means that we need bytecode compiled with source code line information. This is generally the case, even in production applications as it provides extra information on failure events. In the previously referred Map structure, each key corresponds to a given source code point and has a set of associated valid/expected invariant commands (i.e., at a given point there might more than one valid commands, depending on the application).

An important aspect is that the workload should be generated in such way that guarantees a minimum level of code coverage. Coverage analysis tools can be used to measure the workload coverage (we used Cobertura [3] in our experiments). This allows getting information about the coverage space of the workload before deploying the system. Although this does not assure a complete learning of SQL commands and XPath expressions, it allows the developer to have a higher degree of confidence. Obviously, increasing the size of the workload, or choosing a more representative one (e.g., a real application), is a way of improving coverage and further guaranteeing a more complete learning.

C. Service Protection

Service protection at runtime (i.e., after deployment) consists of performing one security check per each data access. Execution is allowed to proceed when that check concludes that the statement is secure, thus presenting no harm to the application or supporting infrastructure. For all other cases, the default behavior is to abort the execution, signaling an exception. Nonetheless, this behavior can be overridden by configuration and an additional check (based on heuristics) can then be executed by a Filter component. These behaviors are explained in the following paragraphs.

During the protection phase, all SQL and XPath commands are intercepted and parsed into invariant codes. The request flow is very similar to the learning phase; the difference is that each request is now delivered to the protector module (see Figure 2). Obviously, the calculated codes are not added to the learned command set. Instead, they are compared to the learned and valid invariant commands for the code point at which the command was submitted.

This matching process consists of looking up the current source code origin in the previously referred Map structure and getting the list of command codes of the valid (learned) commands for that point. This list (generally small) is then searched for an element that exactly matches the invariant command that is being executed. Execution is allowed to proceed if a match is found. Otherwise, a security exception (the unqualified name for this exception is SecurityRuntimeException) is thrown and, in this way, code execution is kept from proceeding, preventing the potential attack. If the source code origin is not found in the Map lookup, execution is also kept from proceeding in a similar manner (in this case, a different exception is thrown – CodePointNotTrainedRuntimeException). This case strongly indicates that the learning phase is incomplete (test coverage was not good enough) and that an extended workload is probably required. We also provide checked versions of these exceptions, for the cases the developer wishes to explicitly state that a web service may throw a particular exception.

As the developer may have no way of verifying the completeness of the training mode (e.g., when there is no access to source code), it may happen that, at runtime, some genuine (i.e., valid) statements are marked for abortion. In this case, and depending on the rate of false-positives (i.e., valid commands marked as invalid), a system administrator has the following options: 1) switch back to training mode (when in a secure environment); 2) verify the tool’s logs and add any false positive to a list of valid commands; and 3) configure the heuristics-based filter for automatic false-positive handling.

The filter uses an attack dictionary to decide if a new (i.e., not yet learned) SQL/XPath statement is potentially malicious (or not). In the positive cases, an exception is thrown (FilteringRuntimeException) stopping any possible damage ahead of time. An excerpt of this dictionary, which can be found packaged with our security tool at [15], is presented in Figure 3.

![Figure 2. Configuration for statements learning and service protection.](image)

![Figure 3. Expressions for well-known SQL attacks.](image)
The dictionary used by the filter is essentially a set of regular expressions and was built based on: current security studies [2][10]; a compilation of attacks generated by well known top commercial vulnerability scanners (Acunetix Web Vulnerability Scanner [1], HP WebInspect [8], and IBM Rational AppScan [9]); and a set of malicious request patterns built by a security team that was challenged to attack a typical web services scenario (see Section IV for a more detailed description of this scenario, team, and experiments). The goal was to obtain as much diversity as possible in the definition of the regular expressions.

An unknown SQL/XPath command can appear either at trained or untrained code points. Considering this, we have designed the following configurable strategies for the filter and protection mechanism:

- **Level 1**: No unlearned SQL/XPath statements are allowed to execute. At runtime, if a data access statement converts into a previously unseen code, then execution is immediately blocked without further analysis (i.e., not delivered to the filter).
- **Level 2**: SQL/XPath statements originating from new (previously unseen) code points are allowed to execute. New SQL/XPath statements originating from old (previously seen) code points are delivered to the filter for further analysis. At the filter, if the data access statement matches with one of the known attack patterns, execution is blocked. If not, execution is allowed to proceed.
- **Level 3**: All unlearned SQL/XPath statements are delivered to the filter for further analysis.
- **Level 4**: All unlearned SQL/XPath statements are accepted for execution. This is essentially an easy way to disable the protection at runtime. It can be used if a severe unexpected case appears (e.g., a serious problem discovered in the learning phase).

Table I summarizes these 4 configurable strategies that add flexibility to the handling of new invariant commands of data access expressions.

<table>
<thead>
<tr>
<th>Level</th>
<th>Code point</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>New/Old</td>
<td>Block</td>
</tr>
<tr>
<td>2</td>
<td>New</td>
<td>Allow</td>
</tr>
<tr>
<td>3</td>
<td>Old</td>
<td>Filter</td>
</tr>
<tr>
<td>4</td>
<td>New/Old</td>
<td>Allow</td>
</tr>
</tbody>
</table>

Obviously, no strategy is able to fit all scenarios, and it is up to the provider to select the strategy that best fits his environment. As mentioned, filter strategies can be changed at runtime. Thus, if the environment changes (e.g., when the service is delivered to a greater number of clients or to a potentially malicious set of clients), the provider can adapt its strategy according to the specificities of that environment.

Before being deployed in a production environment, the protected web service should be assessed in terms of security by using, for instance, vulnerability scanners or manually crafted malicious requests. The goal is to verify if the security mechanism is working properly and to check if our mechanism is able to stop all XPath/SQL injection attempts by raising the appropriate security exception. If vulnerabilities are detected it means that the workload coverage was not good enough and that the learning phase is incomplete. In this case, the workload should be extended and the learning process repeated.

Finally, the developer may also want to re-execute the original workload to verify that the service behavior remains correct. Problem indicators include responses outside the expected domains. For certain services, responses that are different from those obtained during the first workload execution are also problem indicators. These might indicate potential problems introduced by the security mechanism (e.g., due to an incomplete learning of SQL and XPath commands). The process should obviously be canceled if these problems are identified, and their source should be investigated. If this source is related to the learning phase, the developer should extend the workload in order to improve its completeness.

IV. EXPERIMENTAL EVALUATION

In this section we present and discuss the experimental evaluation performed over an initial prototype tool (available at [15]) created to demonstrate the feasibility of the proposed approach. All implementation efforts used Java as a programming language. However, other languages could have been used as well (e.g. C#, C++).

To demonstrate our approach we have used four open-source web services (with a total of 20 operations) publicly available on the Internet [17], and four TPC-App web services (with one operation each) [23]. The public services perform the following functions: manage student information; manage phone book addresses; and simulate bank operations (in 2 versions). TPC-App is a performance benchmark for web services and application servers widely accepted as representative of real environments. The following services were used: Change Payment Method, New Customer, New Product, and Product Detail. These were implemented by a developer with more than 2 years of experience in service-based multi-tier applications (including database access layers).

We chose JBoss 4.2.3.GA as service container and the reference implementation for the Java API for XML Web Services (JAX-WS) due to their relevance in industry [18], [22]. The setup consisted of two nodes (client and server) that were deployed on two machines connected over an isolated Fast Ethernet network.

A. Services Assessment

The first step of the experimental evaluation consisted of trying to identify potential vulnerabilities in the web services. This information was used later to verify the effectiveness of the proposed protection scheme by re-assessing the protected services.
Four well-known commercial vulnerability scanners were used to test the services for vulnerabilities: Acunetix Web Vulnerability Scanner [1]; IBM Rational AppScan [9]; and HP WebInspect [8]. The results were collected exactly as indicated by each scanner and double-checked manually (to discard potential false-positives). Due to scanner usage restrictions, we will refer to these scanners, from this point onwards, as VS1, VS2, and VS3, in no particular order.

The use of vulnerability scanners in this phase enabled us with an initial characterization of the web services. However, it is well known that even the best top current scanners are unable to present accurate results [12], so we asked a team of security experts to perform a code inspection and execute penetration tests to detect additional vulnerabilities. The security analysis team was composed of 5 elements. Three of these elements are developers with more than 2 years of experience on developing database centric business critical web applications in Java. The remaining two are security researchers, one junior (one year of experience) and one senior (four years working on security related topics). The team’s results represent the union of the vulnerabilities detected by each team member. In this case, one vulnerability was counted for each web service input parameter used in a given SQL statement in a vulnerable way. It is important to mention that, as before, we double-checked the vulnerabilities pointed out by each participant (under the form of an example service request) to discard potential false-positives.

Table II summarizes the results and includes a generic service description (total lines of code (LoC) per service and average cyclomatic complexity (Avg. C.), as reported by SourceMonitor – campwoodsw.com). False positives are indicated between parentheses. These detected problems correspond entirely to SQL injection issues, as both the TPC-App specification and the public services do not include any XPath usage. However, the approach is essentially the same; the main difference is each language’s syntax.

<table>
<thead>
<tr>
<th>Web Service</th>
<th>LoC</th>
<th>Avg. C.</th>
<th>VS1</th>
<th>VS2</th>
<th>VS3</th>
<th>Experts</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-App</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChangePayment</td>
<td>97</td>
<td>11.0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2 (2fp)</td>
</tr>
<tr>
<td>NewCustomer</td>
<td>184</td>
<td>9.0</td>
<td>15</td>
<td>15</td>
<td>2</td>
<td>19 (1fp)</td>
</tr>
<tr>
<td>NewProducts</td>
<td>136</td>
<td>6.0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1 (1fp)</td>
</tr>
<tr>
<td>ProductDetail</td>
<td>105</td>
<td>6.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JamesSmith</td>
<td>270</td>
<td>6.0</td>
<td>3</td>
<td>5fp</td>
<td>(1fp)</td>
<td>0</td>
</tr>
<tr>
<td>PhoneDir</td>
<td>132</td>
<td>2.8</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Bank</td>
<td>175</td>
<td>3.4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Bank3</td>
<td>377</td>
<td>9.0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

Concerning the scanners, it is possible to verify that all services, with the exception of one (ProductDetail), presented some kind of vulnerability (the newCustomer service was the service with more disclosed vulnerabilities). Considering the code inspection results, we can see that 3 of the TPC-App services were vulnerable and one in particular had 19 security flaws. This value is essentially due to a large number of user input parameters, being used in more than one SQL statement throughout the code. A large count of vulnerabilities was also obtained for the JamesSmith and Bank3 services. These services use a large number of input parameters, include multiple operations, and some of their operations execute more than one SQL statement in a vulnerable way, which justifies the large number of vulnerabilities found in these two cases.

### B. Statement Learning

The services XML Schema files were manually extended to include domain restrictions (that fully respect the services specifications) for each input and output parameter. EDEL was applied to express the final domains. The workload was then defined and the corresponding coverage analyzed using Cobertura [3]. Table III presents the source code coverage results.

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-App</td>
<td></td>
</tr>
<tr>
<td>ChangePayment</td>
<td>92%</td>
</tr>
<tr>
<td>NewCustomer</td>
<td>80%</td>
</tr>
<tr>
<td>NewProducts</td>
<td>74%</td>
</tr>
<tr>
<td>ProductDetail</td>
<td>94%</td>
</tr>
<tr>
<td>Public</td>
<td></td>
</tr>
<tr>
<td>JamesSmith</td>
<td>90%</td>
</tr>
<tr>
<td>PhoneDir</td>
<td>76%</td>
</tr>
<tr>
<td>Bank</td>
<td>83%</td>
</tr>
<tr>
<td>Bank3</td>
<td>75%</td>
</tr>
</tbody>
</table>

As we can see, coverage is in general above 70%, a value we found to be sufficient for these services. In fact, we decided to analyze the source code of all versions to understand what code was not being covered. In all cases it corresponded to unused exception catch blocks, validation code, or simply code lines that had no influence on the total number of data access statements that could be reached. So, as our simple workload was able to exercise the useful source code perfectly, including all data access statements, and was not expected to trigger any error-handling blocks, we considered the workload adequate for all services.

The workload was then applied to exercise each service operation in order to learn the expected SQL commands. After the learning process, we manually checked whether all possible SQL commands executed by the service application were correctly learned by our mechanism, and that was effectively the case. Note that, the learning process is quite important in our approach and is directly influenced by the coverage of the workload used. If there were commands not learned in this phase we would have to increase the size (and coverage) of the workload. The learning process was quite fast taking only a few minutes and resulting in a total of 16 and 24 invariant data access expressions for the TPC-App and public web services, respectively.
C. Improving Security

After the learning phase, we configured our mechanism to enter the protective state and detect malicious commands. The 3 vulnerability scanners were then used to re-test all services for security vulnerabilities. The results were a total zero disclosed SQL/XPath injection vulnerabilities for all services. For example, in the first assessment, the Change Payment Method service presented a vulnerability when one of the scanners replaced a particular parameter with (‘) resulting in a ‘quoted string not properly terminated’ database error message. With our protection mechanism in place, this type of request corresponds to the generation of a new checksum not detected in the learning phase. This and all new malicious requests were stopped, preventing any further service execution and possible security consequences.

We then replayed all malicious requests crafted by our code inspection participants. All attempts to inject SQL code were aborted by our mechanism by throwing the SecurityRuntimeException exception. Note that our mechanism was not specifically tuned to stop any particular attack. The success of this mechanism is completely related with the process of converting multiple genuine requests executed during the learning phase into invariant commands, thus being fully prepared to detect any abnormal request after the learning phase.

To evaluate the performance of the filtering mechanism, we ran an additional set of experiments. We disabled the invariant determination mechanism and maintained only the regular expression filter. We then re-executed the security scanners over the services. Table IV presents the number of attacks (including multiple variants for a single service parameter) defended by the filter (column ‘defended attacks’) with respect to the total requests originally marked as successful by each scanner (in column ‘tested attacks’).

<table>
<thead>
<tr>
<th>Scanner</th>
<th>Tested Att.</th>
<th>Defended Att.</th>
<th>Defense Success %</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS1</td>
<td>59</td>
<td>49</td>
<td>83%</td>
</tr>
<tr>
<td>VS2</td>
<td>77</td>
<td>52</td>
<td>68%</td>
</tr>
<tr>
<td>VS3</td>
<td>10</td>
<td>10</td>
<td>100%</td>
</tr>
<tr>
<td>Total</td>
<td>146</td>
<td>111</td>
<td>76%</td>
</tr>
</tbody>
</table>

We found that 76% of the attacks originally marked as successful by each scanner (i.e., when testing the services without any protection mechanism) were effectively blocked by our filter. This represents an extremely solid behavior for a generic filter that is only used as a second barrier.

Although our mechanism showed an exceptional performance, we still wanted to check if it could accidentally trigger a security alarm when in presence of genuine accesses. So, we deployed a trained version of the TPC-App services and ran a client emulator (this client is part of the TPC-App specification and was not used in any other part of the experiments) during 60 minutes to see if the security mechanism would trigger a false-positive (the invariant determination or filter components reporting an attack when in presence of a genuine statement). During this process 26092 different data access statements were executed and no false-positives were detected, which is obviously important for users that do not want application’s functionality to break due to any kind of security procedure.

D. Service Behavior Verification

To verify if our mechanism changed the services functionality, we re-submitted the workload to our protected application. The services responses were analyzed to identify deviations from the valid output domains. As expected, no problem was identified, providing a strong indicator that our framework did not change the application’s normal behavior.

To assess the performance impact of the invariant conversion and filtering processes, a final test was conducted. As we were expecting small values, we tested the worst case scenario found in the TPC-App services – the service with more invariant statements to check and the data access expression matching only the last element of the learned statements. We executed 500000 invocations using that worst-case scenario, and the mechanism took on average 0,190 ms (± 0,075) to execute, less than 0,11% of the total time for the fastest executing service. To obtain such low measurements we used a Java method that provides nanosecond precision (but however does not guarantee ns. accuracy).

In summary, the mechanism was able to stop all security attacks with a negligible overhead. This is a very significant result, as besides effectively securing the target application, it implied absolutely no extra-effort from the developers that implemented the services.

V. Conclusion

This paper presented an approach to improve the security of web services. We justify its need with the fact that web services are in general being deployed with various security issues. The approach consists of learning patterns of genuine requests during a training period and using that information to later prevent the execution of malicious and potentially dangerous requests. An additional filter is used to handle new cases arising from incomplete training phases.

Our approach showed to be totally effective in securing a set of open-source services and TPC-App web services against SQL/XPath Injection attacks. This methodology does not require source code access, and as such, during the experimental phase, we did not add any kind of complexity to the services code. This is very important for protecting legacy services, where no source code is available, and results in a helpful tool for developers and service administrators.

ACKNOWLEDGMENT

This work has been partially supported by the project “CRITICAL Software Technology for an Evolutionary Partnership” (CRITICAL-STEP, http://www.criticalstep.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the EU Seventh Framework Programme (FP7).
REFERENCES


