Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services

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

Abstract

This paper proposes a new automatic approach for the detection of SQL Injection and XPath Injection vulnerabilities, two of the most common and most critical types of vulnerabilities in web services. Although there are tools that allow testing web applications against security vulnerabilities, previous research shows that the effectiveness of those tools in web services environments is very poor. In our approach a representative workload is used to exercise the web service and a large set of SQL/XPath Injection attacks are applied to disclose vulnerabilities. Vulnerabilities are detected by comparing the structure of the SQL/XPath commands issued in the presence of attacks to the ones previously learned when running the workload in the absence of attacks. Experimental evaluation shows that our approach performs much better than known tools (including commercial ones), achieving extremely high detection coverage while maintaining the false positives rate very low.

1. Introduction

The security of web applications is, in general, quite poor [5][23]. To prevent vulnerabilities, developers should apply coding best practices, perform security reviews of the code, execute penetration tests, use code vulnerability analyzers, etc. However, many times, developers focus on the implementation of functionalities and on satisfying the user's requirements (and time-to-market constraints), and disregard security aspects.

Web services are nowadays a strategic mean for data exchange and systems integration as they provide a simple interface between a provider and a consumer [6]. However, web services are so widely exposed that any existing security vulnerability will most probably be uncovered and exploited by hackers.

Command injection attacks are very frequent in the web environment [5]. 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 [21], as these frequently use a data persistence solution [19] based either in a relational database or in a XML database.

Different techniques for the identification of security vulnerabilities have been proposed [18]. Penetration testing is a technique in which the testing tool stresses the application from the point of view of the attacker ("black-box" approach) and tries to penetrate it by issuing a huge amount of interactions. On the other hand, static code analysis is a "white-box" approach that consists of analyzing the source code of the application looking for potential vulnerabilities (among other types of defects).

Both penetration testing and static code analysis have advantages and disadvantages. Although penetration testing is based on the effective execution of the code (i.e., provides a runtime view of the web service behavior), vulnerabilities detection is based essentially on the analysis of the web services responses, which limits the visibility on the internal behavior of the code. On the other hand, static code analysis is based on the analysis of the source code (or the bytecode in the case of more advanced analyzers), which allows identifying specific code patterns that are prone to security vulnerabilities. However, it lacks a dynamic view of the real behavior of the service in the presence of a realistic workload (e.g., it does not find vulnerabilities introduced in the runtime environment). Obviously, both providers and consumers can use penetration testing, but only providers can apply static analysis, as it requires access to the code of the service.

In this paper we propose a new automatic approach for the detection of SQL and XPath injection vulnerabilities in web services code. Our approach is based in two main steps. First we generate and run a workload to exercise the web service and learn the profile of the SQL/XPath commands issued. Afterwards we apply a set of command injection attacks and observe the SQL/XPath commands being executed. This allows us to detect existing vulnerabilities by matching incoming commands during attacks with the valid set of commands previously learned.

To demonstrate the effectiveness of our approach we implemented a prototype tool and used it to identify SQL and XPath Injection vulnerabilities in 9 services with multiple operations. Results show that our approach performs much better than existing penetration testing and static code analysis tools. In the experiments performed, it detected 100% of the known vulnerabilities (i.e., the vulnerabilities previously found by a team of security experts), while reporting no false positives.

The structure of this paper is as follows. Section 2 presents background and related work. Section 3 presents the approach proposed. Section 4 describes the experimental evaluation and Section 5 discusses the results. Section 6 concludes the paper.

978-0-7695-3811-2/09 $26.00 © 2009 IEEE
DOI 10.1109/SCC.2009.23

2. Background and related work

In a web services environment, calls between consumers and providers consist of messages that follow the SOAP protocol, which form the core of the web services technology [6], together with WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery, and Integration) specifications. A web service may include several operations and is described using WSDL, which is a XML file, used to generate server and client code. A broker enables applications to find services.

Web vulnerability scanners are well-known penetration testing tools that allow checking applications against security issues. These tools provide an automatic way to search for vulnerabilities avoiding the repetitive and tedious task of doing hundreds or even thousands of tests by hand for each vulnerability type. See Section 4.2 for examples on existing vulnerability scanners.

Previous research shows that the effectiveness of vulnerability scanners on detecting vulnerabilities in web services is very poor. In [21] authors used four commercial scanners (including two different versions of a given brand) to identify security flaws in 300 publicly available web services. The differences in the vulnerabilities detected by each scanner, the low coverage (less than 20% for two of the scanners), and the high number of false positives (35% and 40% in two cases) observed, highlight the limitations of these tools.

Static code analyzers analyze the code without actually executing it [18]. The analysis performed by existing tools varies depending on their sophistication, ranging from tools that consider only individual statements and declarations to others that consider the complete code. Among other usage (e.g., model checking and data flow analysis), these tools provide an automatic way for highlighting possible coding errors. Research on static code analysis is nowadays very intense, including on the identification of the code patterns that are prone to vulnerabilities like SQL and XPath Injection [14]. In Section 4.3 we provide examples of existing static analyzers.

In [13] authors used a *static code analysis* tool for disclosing SQL Injection vulnerabilities in two different versions of the web services specified by the TPC-App performance benchmark [19]. The FindBugs tool has been used, as it is widely used in practice, is able to perform a thorough code analysis, is very easy to use, and has a specific module for SQL Injection vulnerabilities. However, results were quite disappointing. In fact, FindBugs was able to mark only 2 in a universe of 22 vulnerabilities that were previously identified by a team of security experts by executing code inspection and penetration tests.

AMNESIA (Analysis and Monitoring for NEutralizing SQL-Injection Attacks) [8] 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 build a model of the legitimate queries that an application can generate. At runtime, when a query that violates the model is detected, it is classified as an attack and is prevented from accessing the database. Our approach learns the profile of legitimate queries at runtime, which may represent a richer, more realistic learning profile, overcoming the intrinsic limitations of static analysis (e.g., requiring access to source code).

In [4] it is proposed a technique to avoid SQL Injection attacks by comparing, in runtime, the parse tree of the SQL statement before inclusion of user input with that resulting after inclusion of input. The problem is that this technique depends on changes in the source code and, consequently, on its adoption by the programmers.

In [20] it is presented an Intrusion Detection System (IDS) for SQL injection attacks, which intercepts SQL commands (during network communication) and uses several detection models for the different types of attacks. The focus of this work is on the detection of attacks and not on the detection of vulnerabilities.

3. Detecting SQL/XPath Injection vulnerabilities in web services

The approach is based on five steps (see also Figure 1):

1. **Instrument the web service** to intercept all SQL/XPath commands executed.
2. **Generate a workload** based on the web service operations, parameters, data types, and input domains.
3. Execute the workload to **learn SQL commands and XPath queries** issued by the service.
4. **Generate an attackload** (i.e., set of web service calls that include malicious values) based on a large set of SQL Injection and XPath Injection attacks.
5. Execute the attackload to **detect vulnerabilities** by comparing the SQL and XPath commands executed with the valid ones previously learned.

A tool named CIVS-WS (Command Injection Vulnerability Scanner for Web Services) has been designed and built to fully implement the five steps of the approach.

3.1. Web service instrumentation

For the web service instrumentation we use the well-

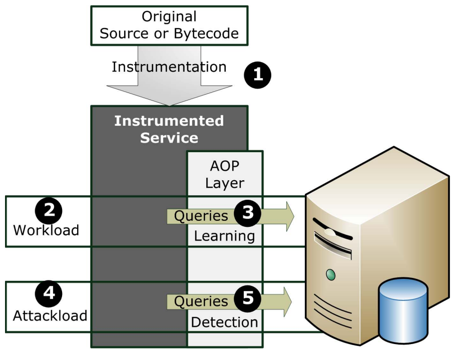


Figure 1. Steps for the detection of vulnerabilities.

known Aspect-Oriented Programming (AOP) paradigm, which allows injecting crosscutting concerns into any application in a non-intrusive way [12]. In practice, we use AOP to intercept key web service execution points and introduce the vulnerability detection mechanisms.

Vulnerability detection starts by automatically identifying all the locations in the web service code where the SQL and XPath commands are executed. This is achieved by using AOP to intercept all the calls to methods that belong to APIs used to execute SQL commands (e.g., Java's JDBC, the Spring Framework JDBC, etc) or to evaluate XPath expressions (e.g., Java's JAXP, JaxenXPath, etc). Besides these well-known APIs, virtually any API can be added, as the only requirement is to know the signature of the method to be intercepted. Figure 2 represents the basic architecture of the interception mechanism.

At runtime, methods that issue SQL commands or XPath queries are intercepted (using AOP) and delivered to a dispatcher. The decision here is simply to check if the application is in learning mode or in detection mode and to deliver the request to the appropriate module (i.e., the learner or detector modules). It is important to emphasize that instrumentation does not change the web service behavior (the code logic is not modified) and that it is only meant for the CIVS-WS tool (it is removed after testing).

3.2. Workload generation

As it is not possible to propose a generic workload that fits all web services, we need to generate a specific workload for each specific service. To generate the workload we need to obtain some definitions about the web service operations, parameters, data types, and domains. As mentioned before, a web service interface is described as a WSDL file. This file is processed automatically to obtain the list of operations, parameters (including return values) and associated data types. However, in most cases the valid values for each parameter (i.e., the domain restrictions of the parameter) are not available in the WSDL and associated schemas. This way, the user is allowed to provide additional information about the valid domains for each parameter. The workload generation is based in the following automatic steps:

1. **Generate test values for each input parameter**: using the definitions mentioned above, the tool randomly generates a set of valid input values (i.e., values in the valid parameter domain). The number of test values to be generated is defined by the user.
2. **Generate test calls for each operation**: the tool creates a large set of calls for each operation. This consists in the sum of all combinations of the test values generated for all the parameters of each operation. For example, for an operation with 5 parameters (p) and 10 test values (v) for each parameter, the total number of calls generated is 9765625 (i.e., p^v).
3. **Select test calls for each operation**: as it may be unfeasible to use a workload based on all the test calls generated (e.g., due to time constraints), the tool is able to randomly select a subset of the calls. Again, it is up to the user to specify the size of this subset, which determines the final size of the workload to be used during the learning step.

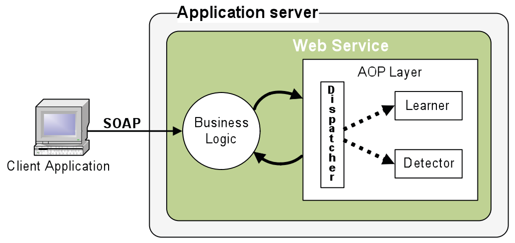


Figure 2. The AOP-based configuration.

3.3. SQL/XPath commands learning

To learn the commands, we exercise the service by executing the generated workload. SQL and XPath commands are intercepted (using AOP) and parsed in order to remove the data variant part (if any) and a hash code is generated to uniquely identify each command. 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 the salary in the select criteria (job like STRING? and sal > NUMBER?) are dependent on the user's choices. Thus, instead of considering the full command text, we just represent the invariant part of it.

Each hash signature is associated with a source code entry point (which is provided by the AOP framework) in a Java Map structure. This does not mean that we need the original application's source code; it just means that we need bytecode compiled with source code line information, which 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 hashed commands. Note that, in a given point there might be several valid commands. For example, as shown in Figure 3, the SQL command submitted to the database might be an insert or an update. This is why we need a set of valid SQL or XPath commands for each source code point.

An important aspect is that the workload must be generated in such way that guarantees a minimum level of code coverage (as discussed in Section 3.2). Although this does not assure a complete learning of SQL commands and XPath expressions, it allows us to have a high confidence degree. Obviously, increasing the size of the workload is a way of improving coverage and further guaranteeing a more complete learning.

```
...
if (isInsert()) {
  sql = “INSERT INTO CLIENT VALUES (seq.nextval, Jack')”;
} else {
  sql = “UPDATE CLIENT SET NAME='John' WHERE ID=1”;
}
statement.execute(sql);
...
```

Figure 3. Example of SQL commands execution.

3.4. Attackload generation

To exercise the web service in the presence of attacks, we generate a malicious workload (i.e., attackload) that includes parameter values that attempt to perform SQL/XPath injection. The set of attack types considered is based on the compilation of the types used by a large set of scanners (three commercial and two open source). This list was complemented based on practical experience and on information on injection methods available in the literature. The final list includes 137 attack types (see Table 1 for examples and [2] for the complete list).

In practice, the first step of the attackload generation consists of generating a new workload by following the steps already presented in Section 3.2. The generation of the attackload based on a new workload is important to avoid exercising, during detection, the same parts of the code that were exercised during learning. Otherwise the parts of the code not exercised during learning would also not be exercised during detection (from the point-of-view of the detection process it would be as if those parts do not exist). As will be explained in Section 3.5, the source code locations exercised during detection, but that were not learned, are logged to indicate an incomplete learning phase. This way, we decided to follow the well-established practice of not using the same set for training and for detection (this is also the approach used in data mining, intrusion detection systems, etc).

After the workload generation, malicious values are selectively inserted by applying the attack rules (see Table 1 for examples; new attack types can easily be added to a configuration file). As shown in Figure 4, this includes several phases, where each phase focuses on generating malicious calls that target a given operation of the web service and includes a set of steps. Each step targets a specific parameter of the operation, and comprises several attack sets. An attack set includes attacks to be performed over a given parameter.

Obviously, the number of attacks to be performed can

SQL Injection Attack
" or 1=0 --
" or 1=1 or ""="
' or (EXISTS)
' or uname like '%
' or userid like '%
' or username like '%
char%2839%29%2b%28SELECT
&quot; or 1=1 or &quot;&quot;=&quot;
&apos; or &apos;&apos;=&apos;

Table 1. Examples of SQL Injection attack types.

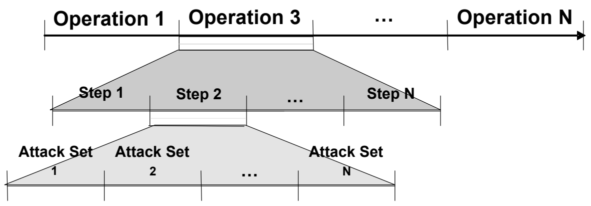


Figure 4. Execution profile for Phase 2.

be extremely huge. Take for example a web service with 3 operations with 5 parameters each and a workload with 25 test calls per operation. Applying all the attack types (137) over all the test calls (25) for every parameter (5) of each operation (3) would end up representing 51375 (137 x 25 x 5 x 3) web service executions. Depending on the time available this might be unfeasible. This way, the tool allows the user to specify the number of test calls that should be used for the attackload generation.

3.5. Vulnerabilities detection

To detect vulnerabilities we execute the attackload and perform security checks per each data access executed. All SQL and XPath commands are intercepted (using AOP) and hashed. The request flow is very similar to the learning phase (see Section 3.3); the difference is that each request is now delivered to the detector module instead of being delivered to the learner module (see Figure 2 for details). Obviously, the calculated hash codes are not added to the learned command set. Instead, they are compared to the values of the learned valid commands for the code point at which the command was submitted.

In practice, the matching process consists in looking up the current source code origin in the previously referred Map structure and getting the set of hash codes of the valid (learned) commands for that point. This set (generally quite small) is then searched for an element that exactly matches the hash of the command that is being executed. If a match is not found, the occurrence (i.e., the potential vulnerability) is logged for future reference. The log entry includes a reference to the code location where the vulnerability was detected, the query that was executed in the presence of the attack, and information about the operation input values, namely the attacked parameter and the attack value. If the source code origin is not found in the Map lookup, the log indicates that the line was not learned. This case indicates that the learning phase is incomplete (coverage was not good enough) and that a more exhaustive workload is required. Note that the lines that have not been learned provide indications on how to improve the workload to increase coverage.

4. Experimental evaluation

An initial prototype tool (available at [2]) was developed 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++).

Experiments were conducted to assess the effectiveness of the approach in detecting vulnerabilities in a set of Java-based web services coded by independent developers. Two key metrics were considered in this evaluation: **detection coverage** (percentage of existing vulnerabilities detected by the tool) and **false positives rate** (percentage of vulnerabilities detected by the tool but that do not exist). A key goal of the experimental evaluation was to compare the proposed approach to other existing tools, including web vulnerability scanners and static analyzers.

4.1. Web services tested

All the operations implemented by 9 web services (in a total of 28 operations) were considered in the experimental evaluation (see Table 2). Four of these services implement a subset of the web services specified by the standard TPC-App performance benchmark [19]. Four other services have been adapted from code publicly available on the Internet [16]. These height services use a database to store data and SQL commands to manage it. This way, we selected and modified 3 of the operations provided by those services to use a XML document as data source and XPath to query the data. These three operations are part of the ninth web service used in the experimental evaluation (the XOperations service). The structure of the code has been maintained as much as possible, and no vulnerabilities have been intentionally inserted or removed from the operations.

Table 2 characterizes the web services (the source code can be found at [2]), including the number of operations per service (#Op), the total lines of code (LoC) per service, the average number of lines of code per operation (LoC/Op), and the average cyclomatic complexity [15] of the operations (Avg. C.). These indicators were calculated using SourceMonitor [17].

4.2. Web vulnerability scanners used

In order to compare our approach to existing penetration testing tools, we selected three well-known commercial vulnerability scanners (representative of the state-of-the-art) and a tool proposed in a previous work [3].

Service		#Op	LoC	LoC/Op	Avg. C.
TPC-App	ProductDetail	1	105	105,0	6,0
	NewProducts	1	136	136,0	6,0
	NewCustomer	1	184	184,0	9,0
	ChangePaymentMethod	1	97	97,0	11,0
Public-Code	JamesSmith	5	270	54,0	6,0
	PhoneDir	5	132	26,4	2,8
	Bank	5	175	35,0	3,4
	Bank3	6	377	62,8	9,0
	XOperations	3	127	42,3	5,3

Table 2. Web services characterization.

HP WebInspect *"delivers fast scanning capabilities, broad security assessment coverage and accurate web application security scanning results"* [9]. IBM Rational AppScan "*is a leading suite of automated Web application security and compliance assessment tools that scan for common application vulnerabilities*" [10]. Acunetix Web Vulnerability Scanner "*is an web application security testing tool that audits your web applications by checking for exploitable hacking vulnerabilities*" [1].

The last penetration-testing tool considered implements the approach proposed at [3]. This tool has shown that, in many cases, it is able to achieve better results than commercial scanners.

For the results presentation we have decided not to mention the brand of the commercial scanners to assure neutrality and because licenses do not allow, in general, the publication of evaluation results. This way, the commercial web vulnerability scanners are referred in the rest of this paper as VS1, VS2, and VS3 (without any order in particular). The fourth tool [3] is referred to as VS.BB.

4.3. Static analyzers used

To compare the effectiveness of the proposed approach with static code analyzers (the most common white-box testing tools), we selected three vastly used tools that provide the capability of detecting vulnerabilities in Java applications' source or bytecode.

FindBugs [7] is *"a program which uses static analysis to look for bugs in Java code"* that is able to scan the bytecode of Java applications detecting, among other problems, security issues (including SQL injection). Yasca [22] is *"a framework for conducting source code analyses"* in a wide range of programming languages, including java. IntelliJ IDEA [11] is a powerful IDE for Java development that includes *"inspection gadgets"* plug-ins with automated code inspection functionalities. IntelliJ IDEA is able to detect security issues, such as SQL problems, in java source code. In the rest of this paper we referred to these tools as SA1, SA2 and SA3.

4.4. Tools setup and configuration

The user-defined values for the CIVS-WS tools were as follows. For each input parameter, 5 values in the valid domain were randomly generated. The workload for the learning process included 100 test calls for each operation. The attackload was generated based on 5 test calls, over which we applied the attack types. This way, the attackload for a given operation included 137 (attack types) x 5 (base test calls) x NUM_OF_PARAMS. For example an operation with 5 parameters, the attackload includes 3425 malicious calls. It is worth noting that the selection of parameters used for the generation of the workload for the learning step and for the generation of the attackload were defined at the beginning. That is, the attackload was not tuned up after the leaning process or based on preliminary detection results. The attackload truly represents an attack scenario that is independent from the learning process.

Before running a penetration testing tool or the CIVS-WS tool over a given service, the underlying database was restored to a predefined state. This avoids the cumulative effect of previous tests and guarantees that all the tools started the service testing in a consistent state. If allowed by the testing tool, information about the domain of each parameter was provided (as was the case of CIVS-WS and VS.BB, which use a similar procedure for the workload generation). If the tool requires the user to set an exemplar invocation per operation (one exemplar is allowed in the case of VS1 and a few in the case of VS2), the exemplar respected the input domains of operation (all the tools in this situation used the same exemplar). Note that, as it uses well-defined input domains, CIVS-WS is able to generate more extensive workloads (and, in principle, more realistic) than the other tools considered.

The static analyzers were configured to fully analyze the services code. For the analyzers that use binary code, the deployment-ready version was used.

5. Results and discussion

This section presents the experimental results. As first step, a team of security experts performed a manual code inspection to identify the existing vulnerabilities. Note that, for the results analysis we assume that the services have no more vulnerabilities than the union of the vulnerabilities detected by the security experts and by the tools. This set of vulnerabilities was then used as reference to evaluate the coverage and false positives of the proposed approach. Other relevant aspect analyzed in the experiments is related to the overhead of using the proposed tool. In fact, the workload generation time was always quite low (a few seconds in all cases). The learning phase was typically of a few minutes per operation (always less than 5 minutes). In all cases, the detection process toke less the 15 minutes per operation.

5.1. Web services manual inspection

To perform a correct evaluation of our approach it is essential to correctly identify the vulnerabilities that exist in the services code. This way, a team of developers with experience in security of database centric applications was invited to review the source code looking for vulnerabilities (false positives were eliminated by cross-checking the vulnerabilities identified by different people).

Penetration-testing tools (that identify vulnerabilities based on the web service responses) count a vulnerability for each **vulnerable parameter** that allows SQL or XPath injection. On the other hand, static analysis tools (that vet services code looking for possible security issues) count each **vulnerable line** in the service code. Due to this dichotomy, we asked the security experts to identify both the parameters and source code lines prone to SQL/XPath Injection attacks.

Table 3 present the summary of the results obtained. The results show a total number of 65 vulnerable inputs and of 32 vulnerable lines of code in the set of services considered. Due to space reasons we do not detail these results. Interested readers can found those at [2].

5.2. CIVS-WS coverage and false positives

The detection methodology of CIVS-WS allows both the identification of vulnerable parameters and of vulnerable lines in the code. CIVS-WS has identified 65 vulnerable inputs and 32 vulnerable lines in the services. All the vulnerabilities detected were compared to the ones identified by the security experts, and we could observe that the tool did not report any non-existing vulnerabilities. This way, in our experiments, the tool achieved perfect coverage (100%) and false positives rate (0%).

We do believe that these excellent results can be generally reproduced in more real world web services. As mentioned before, the quality of the results obtained is directly related to the coverage of the workload used in training phase, as it is during this phase that the valid SQL/XPath commands are learned. The results show that the workload generated by the tool was sufficient to learn the commands in which vulnerabilities exist. This is also possible to achieve in real world situations, as the web services developers typically have access to the source and possess enough knowledge about the service to correctly define the input domains. In fact, the web services from the TPC-App are a good example of real world complex web services (the complexity of the TPC-App specification can be attested in [19]) having a huge number of parameters. Although the TPC-App web services code was not written by the authors of the present paper, it was possible to define a workload that assures a complete learning. This suggests that the process can be reproduced in other real web services scenarios.

Nevertheless, we believe that it is important to research automatic workload generation approaches based on the source code analysis in order to achieve higher code coverage and to increase the tool automation (e.g., to

Service	**Vulnerability Type**	**#Vuln. Inputs**	**#Vuln. Lines**
ProductDetail	SQL Injection	0	0
NewProducts	SQL Injection	1	1
NewCustomer	SQL Injection	15	2
ChangePaymentMethod	SQL Injection	2	1
JamesSmith	SQL Injection	20	5
PhoneDir	SQL Injection	6	4
Bank	SQL Injection	4	3
Bank3	SQL Injection	13	12
XOperations	XPath Injection	4	4
Total		**65**	**32**

Table 3. Vulnerabilities found in the services' code.

eliminate the need for the user to provide input domains).

The *search* operation of the *JamesSmith* web service has a particularity that is important to analyze. This is an operation where the inputs of the operation directly define the structure of the query (ranging from the fields included in *SELECT* clause, to the filters in the WHERE clause, passing by the tables to be queried), and each different combination of inputs causes the query to have a different checksum. In fact, during our experiments we observed that for 640 different combinations of input values, 352 valid checksums were learned. This situation was a real challenge four our tool, and the only way to overcome it was by using a complete workload (with 640 test calls) instead of a reduced workload (i.e., a workload based on a subset of the possible calls). This allowed the tool to learn all the valid queries, guaranteeing an efficient identification of vulnerabilities during detection.

Another interesting situation is related to the service *Bank3*. In this case, the domains of two operations depend on the values previously used by another operation. In practice, the method *newAccount* must be first used to create a limited set of accounts, with a sequential account number (automatically generated) that start at 0. The *deposit* and *withdrawal* operations are then used to update the balance of those accounts. To guarantee a complete learning and consequent success in the detection, this requires that at least some of the test calls generated for the *deposit* and *withdrawal* operations have to use account numbers inside the range of the numbers of the accounts created by the *newAccount* operation. This shows the importance of correctly define the input domains.

5.3. Comparison with penetration testing tools

Figure 5 shows the results for the execution of penetration testing tools and for the CIVS-WS tool. As we can see, the different tools reported different numbers of vulnerabilities. An important observation is that the CIVS-WS tool was able to identify a much higher number of vulnerabilities than the remaining tools. In fact, all the penetration-testing tools detected less than 50% of the vulnerabilities, while our tool detected all vulnerabilities.

Considering only the penetration testing tools, VS1 identified the higher number of vulnerabilities (47% of

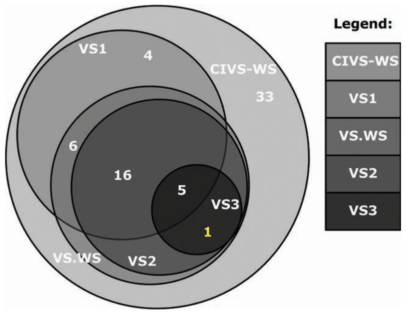


Figure 6. Intersections for penetration testing.

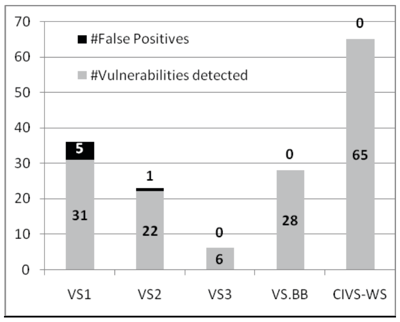
the total vulnerabilities). However, it was also the scanner with the higher number of false positives (it detected 5 vulnerabilities that, in fact, do not exist).

The very low number of vulnerabilities detected by VS3 can be partially explained by the fact that this tool does not allow the user to set any information about input domains, nor it accepts any exemplar request. This means that the tool generates a completely random workload that, most probably, is not able to test the parts of the code that require specific input values in order to be executed.

As different tools detect different sets of vulnerabilities an interesting analysis is how these sets intersect each other. In Figure 6, each circle represents the vulnerabilities detected by a tool and each intersection area represent vulnerabilities found by more than one tool. The circles' area is roughly proportional to the represented number, but the same does not happen with the intersection areas, as it would be impossible to represent it graphically. As we can see, there are 33 vulnerabilities that are detected only by CIVS-WS. Considering only the remaining 32 vulnerabilities, we observe that VS1 misses only one. This is interesting, considering that all the other scanners detected it, although they presented lower coverage than VS1. Additionally, only 5 vulnerabilities were detected by all the testing tools (limited by the low coverage of VS3).

To conclude, our approach has three key advantages when compared to penetration testing:

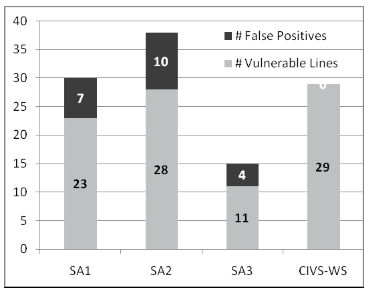
1. Comparing to vulnerability scanners, we generate a more complete workload to exercise the service and learn the SQL and XPath commands issued in the presence of valid inputs.
2. The attacks are a compilation of the attacks performed by a large set of commercial and open source



Tool	% F. P.
VS1	14%
VS2	4%
VS3	0%
VS.BB	0%
CIVS-WS	0%

Tool	Coverage
VS1	48%
VS2	34%
VS3	9%
VS.BB	43%
CIVS-WS	100%

Figure 5. Results for penetration testing.



Tool	Coverage
SA1	71,9%
SA2	87,5%
SA3	34,4%
CIVS-WS	100,0%

Tool	% F. P.
SA1	23,3%
SA2	26,3%
SA3	26,7%
CIVS-WS	0,0%

Figure 7. Results for static code analysis.

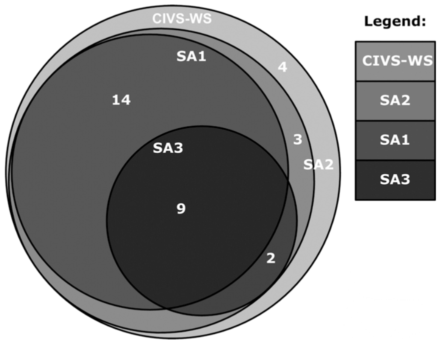


Figure 8. Intersections for static analysis.

web vulnerability scanners plus many methods found in the literature. In other words, we use many more attack patterns than any of the tools tested.

3. We identify vulnerabilities based on the analysis of the XPath and SQL commands actually (to detect the injection of code) and not on the web service response, as is the case of the vulnerability scanners.

5.4. Comparison with static analysis tools

Figure 7 shows the number of vulnerable lines identified by the static code analyzers and by the CIVS-WS tool. As we can see, our tool presents better results than the static analyzers. Considering only the static analyzers, SA2 detected the higher number of vulnerabilities (only missed the four XPath Injection vulnerabilities), with 87.5% of coverage, but identified 10 false positives, which represents 26.3% of the vulnerabilities pointed. The high rate of false positive is, in fact, a problem shared by all the static analyzers used, as all reported more than 23% of false positives. These tools detect certain patterns that usually indicate vulnerabilities, but many times they detect vulnerabilities that do not exist, due to intrinsic limitations of the static profile of the code.

Figure 8 illustrates the intersection of vulnerable lines detected by the different tools. Here it is again visible that CIVS-WS detects all the vulnerabilities found by the static analyzers (excluding the false positives) plus the 4 XPath Injection vulnerabilities. Similarly, SA2 detected the vulnerabilities found by SA1 and SA3, plus other 3 vulnerabilities. Finally, only 9 out of the 32 vulnerabilities were detected by all the static analyzers.

6. Conclusions

This paper proposes a new approach for the detection of SQL and XPath Injection vulnerabilities in web services. The approach is based on XPath and SQL commands learning and posterior detection of vulnerabilities by comparing the structure of the commands issued in the presence of attacks to the ones previously learned. A tool (CIVS-WS) fully implementing the proposed approach has been built, allowing automatic detection of SQL and XPath Injection vulnerabilities in web services.

An experimental evaluation has been performed over 9 web services. Results show that the tool is quite effective, as it was able to achieve 100% coverage and 0% false positives rate. It is worth noting that some of the web services used in the evaluation are very complex, which suggests that the proposed approach can attain very good coverage even for high demanding scenarios.

7. References

[1] Acunetix Web Vulnerability Scanner, 2008, http://www.acunetix.com/vulnerability-scanner/

[2] Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H., “Command Injection Vulnerability Scanner for Web Services”, 2009, http://eden.dei.uc.pt/~mvieira

[3] Antunes, N., Vieira, M., “Detecting SQL Injection Vulnerabilities in Web Services”, 4th Latin-American Symp. on Depend. Computing, João Pessoa, Brasil, September 2009.

[4] Buehrer, G., Weide, B., Sivilotti, P., “Using Parse Tree Validation to Prevent SQL Injection Attacks”, International Workshop on Software Engineering and Middleware, 2005.

[5] Christey, S., Martin, R., ”Vulnerability Type Distributions in CVE”, Mitre report, May, 2007.

[6] Curbera, F. et al., “Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI”, Internet Computing, IEEE, vol. 6, pp. 86-93, 2002.

[7] FindBugs 1.3.8 - http://findbugs.sourceforge.net/

[8] Halfond, W., Orso, A., “Preventing SQL injection attacks using AMNESIA”, 28th Intl Conference on Software engineering, Shanghai, China, 2006.

[9] HP WebInspect, 2008, http://www.hp.com

[10] IBM Rational AppScan, 2008, http://www-01.ibm.com/software/awdtools/appscan/

[11] IntelliJ IDEA 8.1 - http://www.jetbrains.com/idea/

[12] Kiczales, G., et al.: “Aspect-Oriented Programming”, 11th European Conf. on Object-oriented Programming, 1997.

[13] Laranjeiro, N., Vieira, M., Madeira, H., “Protecting Database Centric Web Services against SQL/XPath Injection Attacks”, DEXA 2009, Linz, Austria, September 2009.

[14] Livshits, V., Lam, M., “Finding security vulnerabilities in java applications with static analysis”, 14th USENIX Security Symposium, Baltimore, MD, USA, 2005.

[15] Lyu, M., “Handbook of Software Reliability Engineering”. IEEE Computer Society Press, McGraw-Hill, 1996.

[16] Planet Source Code - http://www.planet-source-code.com/

[17] SourceMonitor 2.5 - http://www.campwoodsw.com/sourcemonitor.html

[18] Stuttard, D., Pinto, M., “The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws”, Wiley, ISBN-10: 0470170778, 2007.

[19] Transaction Processing Performance Council, “TPC Benchmark^{TM} App Standard Spec., Version 1.1”, 2005.

[20] Valeur, F., Mutz, D., Vigna, G., “A Learning-Based Approach to the Detection of SQL Attacks”, DIMVA 2005.

[21] Vieira, M., Antunes, N., Madeira, H., “Using Web Security Scanners to Detect Vulnerabilities in Web Services”, Intl. Conf. on Dependable Systems and Networks, Lisbon, 2009.

[22] Yet Another Source Code Analyzer - http://www.yasca.org/

[23] Zanero, S., Carettoni, L., Zanchetta, M., “Automatic Detection of Web App. Security Flaws”, Black Hat Brief., 2005.