Difficult SQLi Code Patterns for Static Code Analysis Tools

Felix Schuckert[1,2], Basel Katt[2], and Hanno Langweg[1,2]

[1] HTWG Konstanz,
Department of Computer Science,
Konstanz, Germany
felix.schuckert@htwg-konstanz.de
hanno.langweg@htwg-konstanz.de
[2] Department of Information Security and Communication Technology,
Faculty of Information Technology and Electrical Engineering,
NTNU, Norwegian University of Science and Technology,
Gjøvik, Norway
basel.katt@ntnu.no

Abstract

We compared vulnerable and fixed versions of the source code of 50 different PHP open source projects based on CVE reports for SQL injection vulnerabilities. We scanned the source code with commercial and open source tools for static code analysis. Our results show that five current state-of-the-art tools have issues correctly marking vulnerable and safe code. We identify 25 code patterns that are not detected as a vulnerability by at least one of the tools and 6 code patterns that are mistakenly reported as a vulnerability that cannot be confirmed by manual code inspection. Knowledge of the patterns could help vendors of static code analysis tools, and software developers could be instructed to avoid patterns that confuse automated tools.

1 Introduction

Static code analysis tools are commonly used to find vulnerabilities in the development phase of a software project. These tools can be part of continuous integration to report potential security vulnerabilities before those reach a release version. Developers have to review these reports if those reports are actual vulnerabilities. If a reported vulnerability turns out not to be one, the report will produce unnecessary workload. Additionally, if a static code analysis tool does not report an actual vulnerability, vulnerabilities will have a high chance to be included in the release version of the product. Recent research [14] shows that there are source code patterns that are still difficult for static code analysis tools. It is important to identify such difficult patterns to mitigate them in the development phase or improve static code analysis tools to correctly handle them. If such difficult patterns are not known and these patterns are used, software may be developed and deployed with undetected vulnerabilities.

Our contribution is to answer the following research questions regarding SQL injection (SQLi) vulnerabilities:

1. What are difficult source code patterns for static code analysis tools?
2. Is it possible to create simple vulnerability examples with these patterns that are still difficult for static code analysis tools?

We define a difficult source code pattern as a vulnerability pattern that static analysis tools can not identify correctly. This means that either the tools will report it as a vulnerability but it is not (false positive), or they do not report it as a vulnerability but it is (false negative). The term difficult does not represent a metric that includes different difficulties.

Section 2 points out related work. Background and methodology is described in section 3. The static code analysis scan results from the initial open source projects are described in section 4.1. The following section 4.2 shows the results of scanning the minimal working examples. Sections 5 and 6 describe the identified patterns in detail. To see if the patterns are difficult for static code analysis tools a verification is done in section 7. In section 8 we discuss why identifying such patterns is important and what static code analysis tools and developers can do to deal with such difficult patterns.

2 Related Work

Many research projects have evaluated static code analysis tools. Goseva-Popstojanova and Perhinschi [6] evaluated three commercial static code analysis tools. The Juliet database [11] and three open source projects related to Common Vulnerabilities and Exposures (CVE) reports were used to evaluate different permutations of security vulnerabilities. The results show that the tools had high false negative rates and none of the tools detected all vulnerabilities. Our approach also uses source code from open source projects related to CVE reports. Our objective is not to compare the tools with each other, but to instead find difficult source code patterns. Delaitre et al. [4] has similar results. They used as a data set source code from production software that they assumed had no vulnerabilities. Their results show that no tool detected all samples correctly. The data set is similar to ours with both complete open source projects related to CVE reports and specifically created samples. Díaz and Bermejo [5] also evaluated open source and commercial static code analysis tools. They used the test suite 45 and suite 46 from SAMATE [12] database as data sets. For the evaluation, they calculated and compared the F-measure [15] from each tool. AlBreiki and Mahmoud [2] evaluated three open source tools. They evaluated tools with different approaches. OWASP Yasca analyzed source code, FindBugs analyzed byte-code and Microsoft Code Analysis Tool .NET analyzed binary code. They used test cases based on top security issues from OWASP, CWE and SANS. Zhioua et al. [16] evaluated four static code analysis tools based on how they detect vulnerabilities and what techniques are used. They described how security issues and security properties are related to each other. Another approach from Khare et al. [7] evaluated static code analysis tools on large samples with more than 10 million lines of code. The results showed that less than 10% of the vulnerabilities were reported.

Also, software fault injection has an effect on static code analysis tools [3]. Software fault injection actually affects the detection rate of the tested static code analysis tools. Primarily, it leads to false positive reports and existing vulnerabilities were not detected.

There is related work about identifying false positive source code patterns. Reynolds et al. [10] do a more similar approach as ours. They use extract the patterns by manual reviewing them. As test base they are using artificial vulnerabilities from the Juliet framework. Koc et al. [8] are using the previous data set to a classifier to identify problematic patterns. Overall the focus relies on identifying false positive reports. Our results are more focused on patterns that result in false negative reports and our data set is based on open source projects related to CVE reports.

3 Methodology

We use multiple static code analysis tools to find and evaluate problematic source code patterns. Problematic patterns result in false negative and false positive reports. This section provides

background knowledge, explains the chosen commercial and open source static code analysis tools and how the problematic patterns are examined.

3.1 Background

A report from a static code analysis tool can either be true positive (TP), true negative (TN), false positive (FP) or false negative (FN). True positive and true negative reports mean the report is correct and it reported a vulnerability (positive) or it does not contain a vulnerability (negative), respectively. Problematic reports are false negative and false positive reports. A false negative report means that the tool did not report the vulnerability, but a vulnerability actually exists. To find difficult source code patterns that create false negative reports a data set is required that contains vulnerabilities. In contrast, a false positive report means that the tool reported a vulnerability which actually does not exist. To find patterns that create false positive reports a data set is required that does not contain a vulnerability.

3.2 Selected tools

It is not the goal to compare different static code analysis tools to each other. Instead, we want to find source code patterns that are difficult for static code analysis tools. First of all, three commercial static code analysis tools were used. Our licence agreement does not allow to publish the names of the tools. In this work the commercial tools are named Tool A, Tool B and Tool C. All of the tools are state-of-the-art that perform tainted data flow analysis. We focus on static code analysis and in case a tool provides more functionalities we only use the static analysis parts of the tools. Additionally, for the verification phase, two open source tools are used. We evaluated a collection of open source static code analysis tools [1], and selected Exakat (www.exakat.io) and Sonarcloud (https://sonarcloud.io), which are tools for finding security vulnerabilities in PHP and are still maintained (last update $<$ 1 year ago). This approach does not review the internal details and the method that the tools are using in their analysis. As the commercial tools do not allow to examine how they work in detail, the open source tools are also seen as a black box. The goal is to find source code patterns that are difficult for static code analysis tools and reproduce them. On each pattern we describe in detail what problems the static code analysis tools have which prevented them from correctly analyzing the vulnerabilities.

3.3 Data set

The source code patterns should be as realistic as possible. We used a crawler to get source code of open source projects related to CVE reports (www.cve.mitre.org). It uses the categories from CVEDetails (www.cvedetails.com) to filter all CVE reports related to SQL Injection vulnerabilities. These reports are checked for having a confirmation link to a patch on GitHub (www.github.com). The patch itself is checked if it contains any PHP files. CVE reports from 2010 until 2016 were crawled. This ensures that the developers had enough time to patch the vulnerabilities and report the confirmation link to the CVE report. Additionally, all of the samples were manually reviewed to pin point the actually vulnerability. This also takes time and effort to ensure that the correct vulnerability was manually reviewed. Based on that filter criteria we randomly chose 50 CVE reports related to SQL Injection and PHP. We had to limit the number of CVEs considered for this work to be feasible by the researcher at this stage. Expanding the analysis beyond these 50 CVEs can be done in the future. The randomly chosen CVEs are shown in table 5 (appendix). For each CVE report, the developers provided a patch to fix the vulnerability. The source code of the patch is used to create a data set that does not

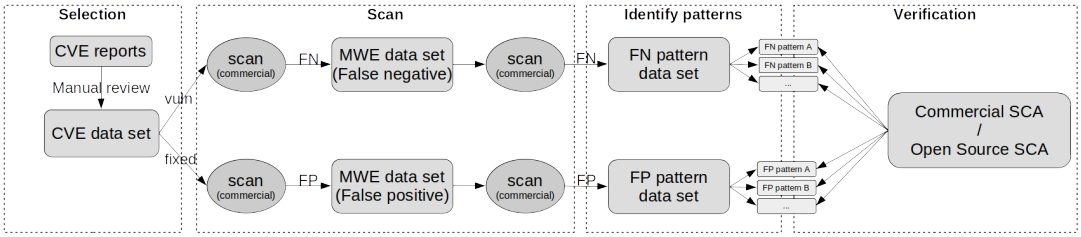


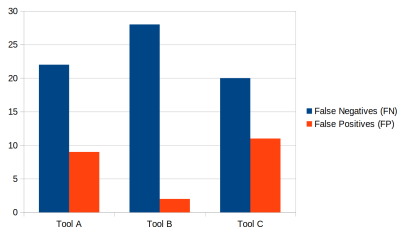
Figure 1: Vulnerability analysis process.

contain the reported vulnerability. This data set are used to find difficult source code patterns that potentially create false positive reports. In contrast, the revision of the source code samples before the patch were used to find source code patterns that might create false negative reports. Both data sets together are further called CVE data set. This data set, after being expanded with recent and more CVEs, can be used as a benchmark for studying and analysing difficult source code patterns.

3.4 Vulnerability analysis

Figure 1 shows the vulnerability analysis process. The process consists of 4 steps, which are (1) selection, (2) scanning, (3) identifying patterns, and (4) verification. As described previously, the CVE data set is split into a data set containing a reported SQL Injection and a data set that patched the vulnerability. All samples are scanned by the commercial static code analysis tools. If a tool does not find the vulnerability, a false negative is identified. In contrast, if a tool reports a vulnerability in the patched version, a false positive is identified. For each false negative and false positive reports, minimal working examples (MWE) were created as follows. First, a basic manual review of the initial source code from the CVE data set was done. This review process is simply tracking the data flow from the related source to the related sink. The identified data flow is then recreated to contain only the related source code. The related source code form the minimal working examples (cf. Figure 1). It can be noted that the goal of creating minimal working examples is to reduce the manual review effort, as its size is much smaller than the actual sample. This MWE data set is scanned by commercial tools again to check, if the problematic patterns are included or not. If a MWE sample is still creating a false negative or false positive reports, the minimal working example is reviewed to identify source code patterns. For each pattern, a sample is created. These samples are created using a PHP file containing a simple SQL injection vulnerability. That file is modified to contain the source code pattern. If the pattern requires multiple files, additional files were added. This creates the next data set named FN pattern data set, which contains patterns that cause false negative reports, and FP pattern data set that cause false positive reports.

The final step in the analysis process is to verify which patterns are actually difficult. To do so, each of the FN/FP pattern data set entries are scanned again. This time the open source tools were used to check if these tools are performing similar to commercial tools. If the sample containing the pattern still creates a false positive or false negative report, a difficult source pattern is identified and confirmed.



(a) FN and FP statistic.

Metric	Tool A	Tool B	Tool C
Vulnerable projects	50	50	50
True Positive	28	22	30
False Negative	22	28	20
Patched projects	50	50	50
True Negative	41	48	39
False Positive	9	2	11
Accuracy	69%	70%	69%
Recall	65%	63%	66%
False Alarm Prob.	24%	8%	26%

(b) Metrics.

Figure 2: Results from the CVE data set.

4 Data set results

As we mentioned before, one of the contributions of the work is the creation of an initial data sets that can be used in the future as a benchmark for difficult source code patterns analysis. Although the data set is limited to 50 randomly chosen CVEs, but this work can be expanded in the future to include more comprehensive and actual material. In this section, we will explain the resulted CVE data sets as well as the minimal working example data set.

4.1 CVE data set results

In order to enable reproducibility of the results of this work, we list in table 5 50 vulnerable and 50 patched projects from the randomly chosen CVE. Figure 2a shows the false negatives and false positives that result from each tool. Even with that few samples (100) it shows a main problem of static code analysis tools. The main problem of static code analysis tools are finding the right balance between false negative and false positive rate. If you want to reduce the false positive rate, it will increase the false negative rate. Table 2b shows an overview of the results and the resulting static code analysis metrics. The accuracy is almost identical on all of the tools. The metrics also show that tool A and C are very similar in all aspects. Tool B is not as good in finding the vulnerability (recall), but it has a very low false alarm probability. Accordingly, if tool B reports a vulnerability, the chance is very high that a vulnerability actually exists. Nevertheless, it did not detect over 50% of the vulnerabilities.

4.2 Minimal working example data set

The minimal working examples were created based on the previous false negative and false positive reports. The minimal working data set can be found on GitHub [13]. Some of the initial data sets result in the same minimal working examples. This happened because some CVE entries were from the same open source project with different versions. In these samples the used source code was the same, so we only created one minimal working example. The minimal working example samples were scanned again to see if the important parts were found. Table 1 shows the results of the minimal working examples that should result in a false negative report from at least one tool. In two samples, all tools detected the vulnerability. Accordingly, we were not able to construct a minimal working example for these samples. Similar to the initial scans, Tool B has the most false negative reports. Tool C performs very well in this data set with only one false negative report. We reviewed the results manually to find the reason for the different results between the CVE data set and MWE data set. Tool C reports a SQL injection

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FN
CVE-2011-4960	ReflectionClass	FN	FN	✓	2
CVE-2012-0973	Get parameter function Singleton Func_get_args Function - sprintf	✓	FN	✓	1
CVE-2012-2762	-	✓	✓	✓	0
CVE-2012-3470	Inerhit query construction	✓	FN	✓	1
CVE-2012-3471	Eventmanager	✓	FN	✓	1
CVE-2012-5162 CVE-2013-3527 CVE-2015-4628 CVE-2016-9020 CVE-2016-9087 CVE-2016-9183 CVE-2016-7453 CVE-2016-9242 CVE-2016-9272 CVE-2016-9282	Get parameter function Database access object	FN	FN	✓	2
CVE-2013-2559	Database - static method Eventmanager	✓	FN	✓	1
CVE-2013-3081	Environment variable	✓	FN	✓	1
CVE-2013-3524	-	✓	✓	✓	0
CVE-2013-4789	Get parameter function	✓	FN	✓	1
CVE-2014-1608 CVE-2014-1609	SOAP	✓	FN	✓	1
CVE-2014-5017 CVE-2016-7400	Sub class get method Singleton - set Singleton - classes	✓	FN	✓	1
CVE-2014-9089	Sanitize only limited elements Explode - implode	✓	FN	✓	1
CVE-2014-9464	Singleton __get __set	✓	FN	✓	1
CVE-2014-9528 CVE-2014-9573	Get parameter function Complex query construction	✓	FN	✓	1
CVE-2015-4426	Stored class Json decode	FN	FN	✓	2
CVE-2016-2555	Database - global variable Sanitize function not initialized	FN	FN	✓	2
CVE-2016-5703	Database - global array Imported variable Imported sink	✓	FN	FN	2
Summary (FN)		4	16	1	

Table 1: Scan results from false negative MWE data set with 18 samples.

vulnerability if a string variable is concatenated that is used in a database sink. If no source is found, it still reports a potential SQL injection with a lower priority.

The results for the false positive data set is seen in table 2. Tool A and Tool C are reporting more false positives. The minimal working example (CVE-2011-4960) did not include the relevant source code patterns to create a false positive report because all tools correctly reported a true negative.

CVE related	Patterns	Tool A	Tool B	Tool C	Count of FP
CVE-2011-4802	Preg_match - check for number	FP	FP	FP	3
CVE-2011-4960	-	✓	✓	✓	0
CVE-2012-2762	Function - strreplace Sanitize if function exists	FP	✓	FP	2
CVE-2013-2559	Sanitize if function exists	FP	✓	FP	2
CVE-2013-4789	White listing	FP	✓	FP	2
CVE-2014-3773 CVE-2012-5162 CVE-2011-4341 CVE-2012-0973 CVE-2015-1471 CVE-2013-4879 CVE-2014-8351	Function - quote	FP	✓	FP	2
CVE-2015-2679	Function - htmlentities	FP	✓	FP	2
CVE-2016-7780	Sanitize function - global db variable	FP	✓	✓	1
Summary (FP)		7	1	6	

Table 2: Scan results from false positive MWE data set.

Figure 3: Overview of source code pattern categories.

5 False negative source code patterns

The minimal working examples commonly contained multiple source code patterns. All the used source code patterns were reviewed and based on that pattern a sample was created containing the pattern only. The source code for each pattern can also be found on GitHub [13]. The patterns are categorized into source, concatenation, sink, sanitization and data flow. Figure 3 shows the categories and how they are found in a typical SQL injection vulnerability. The data flow (DF) patterns are in between other patterns. The different source code patterns are described in this section.

5.1 Sources

This section describes the source code patterns that are a source. All of the following patterns have some special parts that makes it difficult for static code analysis to detect it as a source:

- **Sub class get method**
  This source code pattern uses inheritance. The super class implements a method that uses the *method_exist()* to check if the sub class implemented a get-method. If the getter method is implemented, it will be called using call-method-by-string. Call-method-by-string is an unconventional way of calling a method. Static code analysis tools have to parse the corresponding string that might also require additional data flow analysis. In our results only tool B was able to correctly parse the string to the corresponding method.

- **Get parameter function**
  A common found pattern in our data set was using wrapper methods for getting user data. It simply defines a method that uses common PHP (e.g. *$_GET*) methods to get user data. This pattern includes a sanitization method which is not used by default. Tools have to check, if the sanitization method is enabled or not to correctly detect a vulnerability. The commercial tools were able to correctly detect a SQL vulnerability including this pattern.

- **SOAP**
  PHP allows to implement a Simple Object Access Protocol (SOAP). This pattern uses the *SOAPServer* class to implement a SOAP service. Parameters passed to that service are user data and potential dangerous. The problem for static code analysis tools relies on how such a service will be implemented in PHP. The *SOAPServer* class uses the *setClass* method to register a class name defined by a string. The tools have to backtrack the string value to actually know the class name of the provided SOAP service. If they have correctly backtracked the string value, the tools can mark all method parameters of the class as possible sources.

- **Import**
  A very simple source code pattern (Imported variable) uses an additional PHP file which is imported by the *require_once* function. The imported file simply uses the *$_GET* method to get user data and stores it in a PHP variable. This pattern requires that includes from other PHP files are parsed correctly. The commercial tools have no problem with includes

from other PHP files.
Another variant of that is the Stored class pattern. It is a bit more complex because the included file defines a class that stores tainted data in a class variable. Later on tainted data is retrieved by another get method that uses the class variable. Again the commercial tools have no problem tracking that tainted data is stored in class variables.

5.2 Insufficient sanitization

The false negative patterns are based on source code that contains a SQL Injection vulnerability. Accordingly, the user input was not sanitized correctly. We found the following two patterns that were containing insufficient sanitization methods:

- **Sanitize only limited elements**
  In this pattern user data is provided as an array. The data is iterated and sanitized by a white listing check. Only user data that is in the white list is allowed. The special part of this pattern is that only a limited amount of values are sanitized. In our data set only the first two elements of the array were sanitized. The tools have to check, if all elements in a array are checked. This requires the tool to keep track on how large an array might be. For example, if the array can only be the size of two elements, checking only the first two elements would be sufficient. This pattern sounds artificial, but our CVE data set actually had such a code pattern where only the first two elements were sanitized.

- **Sanitize function not initialized**
  This pattern uses a global string variable which defines the sanitization methods. This allows developers to decide which sanitization method will be used. A default method is already defined which only returns the provided string without any sanitization. The sanitization method is called using dynamic method invocation [9]. Again, this requires the static code analysis tools to parse the corresponding string to see what method is called. If the variable is defined global, it makes it even more difficult. It requires to correctly parse the PHP project. The global variable might be defined multiple times and the correct variable can only be tracked by parsing all the includes of different files. None of our tested tools were able to correctly detect this pattern.

5.3 Concatenation

Patterns in this category are describing how the SQL query is build. It describes the concatenation between the user input and the SQL query. We found following patterns:

- **Complex query construction**
  In this pattern the SQL query is constructed using multiple functions. It uses a class to store the query relevant data. The concatenation of the SQL query is a concatenation of multiple method return values. Each of these method is a construction part of the query. For example, one method creates the *WHERE* clause of the query. The tainted data is stored in class variables. This requires the static code analysis tools to correctly track the tainted data in multiple method calls. Commonly, it includes also other functions from PHP. For example, the *implode* functions was found a lot of times to combine multiple parameters into one query. If only one part is incorrectly analyzed, the vulnerability is not detected.

- **Function - sprintf**
  This pattern uses the function *sprintf* to construct a SQL query in a c-like fashion. It allows to define a string with different specifiers which will be replaced by parameters. This is a common way to concatenate a string with variables. Static code analysis tools have to add the *sprintf* functions and correctly parse the variables to the correct positions in the string value. It also requires the tools to determine between a string replacement or just a fixed value replacement. For example, if only a number is inserted using the *sprintf* function, the tainted data is not tainted anymore. Accordingly, the *sprintf* function could also be used as a sanitization function. In our pattern, we used the string replacement which does not prevent any vulnerabilities.

5.4 Sink

Sinks are critical functions, if user input reaches it without any sanitization in between. Patterns of this category are different implementations for a SQL Injection sink.

- **Database access object**
  This was a very common source code pattern in our data set. A database access object (DAO) is a simple PHP class that stores the query relevant data as object variables. The DAO is able to construct the SQL query string using object methods. Similar to the the concatenation pattern (Complex query construction), the SQL query construction requires to track multiple method calls and tainted data stored in class variables. The main difference is that this pattern also stores the database object as a class variable. Calling corresponding functions on the DAO objects will construct the SQL query and also perform the query on the database. It then just returns the resulting data.
  A special case of this pattern is the (Inerhit query construction) pattern. It also uses a database access object, but relevant implementations are defined by inherited classes. The tools have to know what sub class is used to decide, if a vulnerability exists or not.

- **Database object storage**
  In our CVE data set we found different ways of storing a database object. The connection to the database itself is usually only established once. Then the corresponding object is stored in different ways. It either is stored in a global array (Database - global array). This requires that the static code analysis tools are correctly tracking global arrays and what data is stored. Or it is stored in a global variable (Database - global variable). Another pattern was using static methods to connect to an database and to statically get the corresponding database object (Database - static method).

5.5 Data flow

The data flow is a relevant part of SQL Injection vulnerabilities. The data passes different source code pattern between the source and sink. These patterns are based on SQL Injection vulnerabilities, but data flow source code patterns are also relevant for other vulnerability types.

- **Eventmanager**
  An Eventmanager is used to create a system based on events. The implementation uses the static class methods *add* and *run*. The method *add* allows to add callbacks to specific events. If the *run* method is used, a specific event is run and all related callbacks are called. The callbacks are stored in static class variables. Static code analysis tools have to track all callbacks that are stored in the Eventmanager class. Programs written using an

Eventmanager are completely different than a objective oriented programming style. It is much more difficult to parse all the possibilities of what kind of events occur and it might even be unpredictable. Our pattern sample uses predefined events that always end up in a vulnerability.

- **PHP pass through functions**
  There are multiple PHP functions that returns data from a parameter (pass through). Static code analysis tools have to define these functions as passing through tainted data. In our data set we found following functions *explode*, *implode* and *json_decode*. None of these functions are changing the data in a way that it prevents a SQL injection vulnerability. Accordingly, the tools should define these function as pass through functions.

- **Dynamic PHP functionality**
  PHP provides a lot of language features. It has functionality that allows to dynamically call functions and methods. Our data set showed the usage of *func_get_args* function. This function allows to get function parameters without defining them at the function definition. This makes it very difficult for static code analysis tools to correctly track the parameters. The parameters are returned as an array.
  Also our data set showed the definition of *__get* and *__set* class methods. These methods are called when a class variable is accessed that is not defined in the class definition. In the set method the class variable name and the value are passed as parameters. The get method only has the class variable name as parameter. Our implementation just stores the value for the corresponding class variable name and returns the corresponding value on the get method. Nevertheless, the implementation might differ and static code analysis tools have to analyze the methods. If the methods are analyzed then all corresponding class variable accesses without a class variable definition have to be tracked to correctly analyze the program. Our pattern sample is very simple to see if static code analysis tools are analyzing the dynamic get and set methods.
  Another dynamic feature of PHP is setting and getting environment variables (Environment variable). It provides the function *putenv* to set an environment variable. The parameter type is string. The string itself requires to be in a specific format (*"Varname=Value"*) to actually set an environment variable. The corresponding function *getenv* is used to get the value of an environment variable. The parameter of the function is a string that defines the variable name. Static code analysis tools have to analyze and backtrack the corresponding strings to correctly analyze this pattern.

- **Plugin support**
  This a very complex source code pattern. It actually might be more a architecture, but it is commonly found in our CVE data set. It uses a plugin structure that allows to easily add more modules. In our sample for each plugin a controller and view class has to be implemented. These implementations are sub classes from template classes. These implementations have to be in a subdirectory in a fixed plugin structure. The source code pattern parses the plugin folder for valid plugin implementations. A valid plugin implementation can then be accessed as it would be normal PHP web page. For this access a router class is implemented which routes to the correct plugin. The usage of plugins as a developer is convenient. It allows to split the programs in different modules. In contrast, static code analysis tools have problems analyzing plugin supported programs. First of all the programs has to be analyzed to see that the program itself has plugin support. The analyze also has to find out what files are included from a plugin. Because plugin support

is usually not fixed to a specific number of plugins, the loading off such plugins is dynamic. Loaded plugins are stored in class variables and corresponding PHP files are loaded.

- **Singleton**
  The simple singleton sample is just implementing the common known singleton pattern. Singleton is a source code pattern that allows to access only one instance of an object. The singleton itself has a get method to access the *$_GET* parameter. Because a singleton can be accessed from everywhere, the static code analysis tools have to track all possible ways of calling the singleton. Our sample is just a procedural calling of the singleton.
  Our CVE data set also showed that the singleton pattern was used with different implementations. The Singleton - classes pattern allows to get class objects by a name. The name itself is a string variable that requires the static code analysis tools to actually analyze the string variable to know what class object is returned.
  Another pattern (Singleton - set) returns one instance of an object. The object itself is not predefined. Initial a corresponding set method has to be used to set the singleton object. Afterwards the singleton object can be accessed from everywhere.
  The ReflectionClass pattern is not a singleton per definition. It uses a static class implementation to create new class objects. Accordingly, you can access the class from everywhere, but you will always get a new object. It uses a combination of the PHP functions *func_get_args* and *array_shift* to get the class name and parameters provided as a function parameter. The new instance of the class is created using the *ReflectionClass* from PHP standard library. The class name itself is again a string parameter that has to be analyzed by the static code analysis tools to get the corresponding class of the returned object.

6 False positive source code patterns

Source code patterns in this category are patterns that developers used to fix the reported vulnerabilities. The described source code patterns are sufficient to prevent SQL injections, but still static code analysis tools are reporting a vulnerability.

- **Official sanitization**
  Database driver usually provide a sanitization function to sanitize tainted data. We used the *quote* function provided by the PDO class from PHP. The initial CVE data set also contained old source code samples that used functions like *mysql_real_escape_string*. These functions are not supported anymore and the PHP documentation provides the *quote* as an alternative. Accordingly, we added the *quotes* function as pattern. This function should be added to a static code analysis tool as a sanitization method. As described in next section, this sanitization method is correctly detected by all the tested static code analysis tools.
  Another pattern uses the *htmlentities* function. This is not a specific function to prevent SQL injection vulnerabilities. This source code pattern just uses the *htmlentities* function to sanitize the user data. But it requires that the SQL query itself adds quotes around the sanitized data to ensure that a SQL injection is not possible. Accordingly, static code analysis tools have to analyze the query to see, if the sanitization with *htmlentities* is sufficient or not. This is a problem of many sanitization method, that they are sufficient enough in a specific context. Some of them are sufficient enough for a specific vulnerability type and some of the sanitization methods like the *htmlentities* is only sufficient enough based on the SQL query statement.

- **Custom Sanitization**
  The CVE data set revealed multiple custom sanitization implementations. The function *preg_match* can be used to check a string value based on a regular expression (regex). The regular expression is an important part because based on that expression a SQL injection can be prevented or still might be insufficient. Static code analysis tools have to analyze the regular expression to see, if the sanitization method is sufficient. This pattern uses a regular expression that checks, if the string value only contains numbers. Accordingly, this is sufficient to prevent any SQL injection attacks.
  Another pattern we found, uses the *str_replace* function to replace any dangerous characters. The initial sample replaced all apostrophes and the SQL query itself puts the user data inside apostrophes. This time the static code analysis tools have to the analyze of the regular expression and the SQL statement to determine, if the sanitization method is sufficient or not.

- **White listing**
  White listing is a common way to mitigate any attack. Only fixed inputs are allowed. These fixed inputs should be chosen that they are not creating any attack possibilities. The implementation of white listing can differ. Our implementation based on the CVE data set an array is defined that contains all the allowed inputs. The *isset* function is used to check if the user data is contained in the white list array. Accordingly, static code analysis tools have to analyze the content of the white list array to see what inputs are possible. The creation of the array might be complex. A static code analysis tools also has to check all the possible inputs, if any of these inputs might still result in a vulnerability.

- **Dynamic defined sanitization functions**
  Many of the CVE data set source code samples are from frameworks. They allow to define what kind of database is used. Also some of them allow to define what sanitization method will be used. The Sanitize function - global db variable pattern uses a wrapper method for the *escapeString* function. The wrapper function is implemented as a static class function. The implementation itself uses a global defined variable for accessing the database connection object. This object provides the relevant sanitization method (*escapeString*). This global variable is defined in the initialization process. This makes it very difficult for static code analysis tools because they have to analyze what sanitization function is defined in the initialization process. Based on what function is used, it also may require to analyze the SQL query to see, if it is sufficient.
  The Sanitize if function exists pattern uses a wrapper function for the *quote* function. The wrapper function uses the *function_exists* function to check, if the sanitize function actually exists. In our sample the function exists because it checks for a standard php library. Nevertheless, if an older PHP version is used, it might not exist. This makes the pattern also PHP version dependent. The static code analysis tools also have to know what PHP version is used.

7 Verification

The difficult source code patterns were scanned again to verify what patterns are actually difficult. A simple sample was used which contains a simple SQL injection vulnerability. For each of the previous described patterns the simple sample was modified to contain the pattern. The evaluation also used the open source static code analysis tools. The created source code patterns can be found on GitHub [13].

False negative pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FN
Source						
Sub class get method	FN	✓	FN	FN	FN	4
Get parameter function	✓	✓	✓	FN	FN	2
SOAP	✓	FN	FN	FN	FN	4
Imported variable	✓	✓	✓	FN	FN	2
Stored class	✓	✓	✓	FN	FN	2
Insufficient sanitization						
Sanitize only limited elements	✓	FN	✓	FN	✓	2
Sanitize function not initialized	FN	FN	FN	FN	FN	5
Concatenation						
Complex query construction	✓	FN	✓	FN	✓	2
Function - sprintf	✓	FN	✓	FN	✓	2
Sink						
Database access object	FN	FN	✓	FN	✓	3
Database - global array	✓	✓	✓	✓	FN	1
Database - global variable	✓	✓	✓	✓	FN	1
Database - static method	✓	FN	✓	FN	FN	3
Inerhit query construction	✓	FN	✓	FN	FN	3
Database - wrapper	✓	✓	✓	✓	✓	0
Imported sink	✓	✓	✓	✓	✓	0
Data flow (DF)						
Eventmanager	FN	FN	FN	✓	FN	4
Explode - implode	✓	FN	✓	FN	✓	2
Func_get_args	✓	FN	FN	FN	FN	4
__get __set	✓	✓	FN	FN	✓	2
Json decode	FN	FN	✓	FN	FN	4
Plugin support	✓	FN	FN	FN	FN	4
Environment variable	FN	FN	✓	✓	FN	3
Singleton	✓	✓	✓	FN	FN	2
Singleton - classes	✓	FN	FN	✓	FN	3
Singleton - set	FN	✓	✓	✓	FN	2
ReflectionClass	FN	FN	FN	FN	FN	5
Summary (FN)	7	15	9	19	18	

Table 3: Scan results for different false negative patterns.

Table 3 shows the result for the false negative patterns. Two identified source code patterns were not creating a false negative report. Accordingly, these two patterns (Imported sink, Database - wrapper) are not difficult for state of the art static code analysis tools. The open source tools already have problem with simple patterns like Get parameter function, Imported variable, Stored class and Singleton. The different singleton implementations makes it difficult even for the commercial tools. Interestingly, the open source tools detected some of the singleton implementations correctly. The ReflectionClass pattern was difficult for all of the tested tools. That pattern includes different dynamic PHP features and the combination of creating a class object based on a string value. Accordingly, it includes many already difficult sub patterns. Also the Plugin supportis a pattern that includes different PHP features together to create a plugin support. None of the tools were able to detect them.

Table 4 shows the result for source code patterns related to false positive reports. The Function - quote pattern was not difficult for any of the tested tools. As already state the initial CVE data set also contained old source code using outdated sanitization methods. These methods were actually creating false positive reports because the static code analysis tools did not have them in their list of sanitization functions. It also shows that all of the tested static code analysis tools are checking for sanitization methods. The results show that the tools are very different on detecting sanitization approaches. Some of them are even considering custom sanitization attempts and other tools just ignore them.

False positive pattern	Tool A	Tool B	Tool C	Exakat	Sonarcloud	Count of FP
Sanitization						
Preg_match - check for number	✓	FP	FP	FP	FP	4
Function - quote	✓	✓	✓	✓	✓	0
Sanitize if function exists	FP	✓	FP	✓	FP	3
Sanitize function - global db variable	FP	✓	✓	✓	✓	1
Function - htmlentities	FP	✓	FP	✓	✓	2
Function - strreplace	FP	✓	FP	✓	✓	2
White listing	✓	FP	FP	FP	FP	4
Summary (FP)	4	2	5	2	3	

Table 4: Scan results for different false positive patterns.

8 Discussion

The evaluation shows that almost all of the identified patterns are difficult for at least one static code analysis tool. Some of these patterns are just common programming functionality provided by PHP. Such functionality should not be difficult for modern static code analysis tools. Most patterns are related to SQL injection vulnerabilities, except the patterns in the data flow category. These are patterns transferring user data from one point to another. These patterns are interfering the data flow algorithm from the static code analysis tool. Accordingly, these patterns are not difficult just for SQL injection vulnerabilities, instead they are difficult for all vulnerability types that require user input reaching critical functions. The developers of the static code analysis tools should be able to improve their algorithms to get fewer false negative and false positive reports. Nevertheless, some of the patterns are not that easy to be detected correctly, especially if the patterns contain dynamic language features of PHP. If developers use such a feature they should only use it if it is necessary. As our results show as more of such dynamic features are included, the more false negative reports occur. Also if regular expressions are involved, the tools have to parse the expression. Based on the expression, the sanitization might be sufficient or not. Usually the expression is also related to the SQL query. Our presented patterns can be prevented in the development phase. These patterns can be replaced by code patterns that can be easily detected by static code analysis tools. For example, using the *str_replace* function for replacing critical characters can be replaced by using common known sanitization methods provided by the database library. Static code analysis tools know that these sanitization methods are sufficient to prevent any SQL injection attacks. Accordingly, the tool is then not reporting a false negative report.

This work required different manual review steps. The source code of the different open source projects were initially scanned by the tools. The results had to be reviewed manually to find all the false negative and false positive reports. Because of the manual review process the data set was limited. 50 false negative samples and 50 false positive samples were used to find the previously described difficult source code patterns. Some of the samples were even from the same open source project. Because of the small data set, we can be sure that there are still more difficult source code patterns for static code analysis tools. The different tools also could not be compared to each other because of the small data set. The results show a tendency that commercial tools outperform open source tools. Especially that the difficult source code patterns were identified specifically based on the false negative/false positive reports from the commercial tools. There were a lot more difficult patterns for false negative reports found than for false positive results. The reason for this is that we only reviewed reports based on the patched versions. If a tool did not report the vulnerability in the vulnerable version, the modifications of the patch will not create a false positive report. Finding a solution for the manual review steps would allow to research for difficult source code patterns on a broad scale.

The CVE data set itself does only include reports until end of 2016. The reason is that the manual reviewing of the source code pattern takes a significant amount of time. The resulting patterns are all updated to the up to date PHP version with corresponding functions. The tested static code analysis tools are all state of the art and the pattern are still difficult for them. A newer CVE data set might introduce even more difficult source code patterns. Nevertheless, our results show that the patterns we created are still difficult.

9 Conclusion

The goal to find difficult source code patterns was successfully achieved. The review of 50 open source projects containing vulnerable and patched versions revealed 25 difficult source code patterns for false negative reports and 6 difficult source code patterns for false positive reports. The verification shows that modifying simple vulnerabilities with these patterns are still difficult for static code analysis tools. The dynamic language features of PHP are nice for programmers, but for static code analysis tools they are very difficult. The results show that most identified patterns are data flow patterns. Many patterns should be detected by modern static code analysis tools and their developers should improve the algorithms based on our results. Nevertheless, some patterns can also already be mitigated during the development phase. Developers should know what patterns are difficult for static code analysis tools. Our patterns can be used as learning examples for teaching higher level of software security.

References

[1] PHP SCA tools. https://github.com/exakat/php-static-analysis-tools, 2020.

[2] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, 2014.

[3] T. Basso, P. C. S. Fernandes, M. Jino, and R. Moraes. Analysis of the effect of Java software faults on security vulnerabilities and their detection by commercial web vulnerability scanner tool. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 150–155, 2010.

[4] A. Delaitre, B. Stivalet, E. Fong, and V. Okun. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 14–20, 2015.

[5] G. Díaz and J. R. Bermejo. Static analysis of source code security: Assessment of tools against SAMATE tests. *Information and Software Technology*, 55(8):1462–1476, 2013.

[6] K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

[7] S. Khare, S. Saraswat, and S. Kumar. Static Program Analysis of Large Embedded Code Base: An Experience. *Proceedings of the 4th India Software Engineering Conference 2011*, pages 99–102, 2011.

[8] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM*

Table 5: CVE data set.

CVE	Github link	FN (A)	FN (B)	FN (C)	FP (A)	FP (B)	FP (C)
CVE-2016-9283	https://github.com/exponentcms/exponent-cms/commit/559792be727f4e731bfcb3935f5beec7749e9ce9						
CVE-2016-9282	https://github.com/exponentcms/exponent-cms/commit/e83721a5b9fcc88e1141a8fb29c3d1bd522257c1	x	x	x			
CVE-2016-9272	https://github.com/exponentcms/exponent-cms/commit/ffb2038de4e603931b785a4c3ec69cfd06181ba	x	x	x			
CVE-2016-9242	https://github.com/exponentcms/exponent-cms/commit/6172f67620ac13fc2f4e9d650c61937d48e9ecb9	x	x	x			
CVE-2016-9183	https://github.com/exponentcms/exponent-cms/commit/3b3557e9f6ba193a4c23c8ce5498fa285dddf3f3	x	x	x			
CVE-2016-9134	https://github.com/exponentcms/exponent-cms/commit/45a7a62797e64e8abbac35d4859097c26f1874b1						
CVE-2016-9087	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db	x	x	x			
CVE-2016-9020	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db	x	x	x			
CVE-2016-7788	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db						
CVE-2016-7781	https://github.com/exponentcms/exponent-cms/commit/fdafb5ec97838e4edbd685f587f28d3174ebb3db						
CVE-2016-7780	https://github.com/exponentcms/exponent-cms/commit/a8efd9ca71fc9b8b843ad0910d435d237482ee31				x		
CVE-2016-7453	https://github.com/exponentcms/exponent-cms/commit/c1092f167cc6c78dc8bf9bf149946c5219413df3	x	x				
CVE-2016-7405	https://github.com/ADOdb/ADOdb/commit/bd9eca9f40220f9918ec3cc7ae9ef422b3e448b8						
CVE-2016-7400	https://github.com/exponentcms/exponent-cms/commit/e916702a91a6342bbab483a2be2ba2f11dca3aa3		x				
CVE-2016-5703	https://github.com/phpmyadmin/phpmyadmin/commit/ef6c66dca1b0cb0a1a482477938cfc859d2baee3	x		x			
CVE-2016-2555	https://github.com/atutor/ATutor/commit/945a9dca01def8536516088da30fe6a4b7e9fa85	x	x	x			
CVE-2015-5078	https://github.com/LimeSurvey/LimeSurvey/commit/65d717415a271242b9a30a5330d4eabac1c1a837						
CVE-2015-4628	https://github.com/LimeSurvey/LimeSurvey/commit/b09edc0dbd18d8459ade4c7c941e562c16564f9e	x	x	x			
CVE-2015-4426	https://github.com/pimcore/pimcore/commit/1c6692e8287deed7f3356b6a1e2e9b7fe4e858dd	x	x	x			
CVE-2015-2679	https://github.com/semplon/GeniXCMS/commit/6982454883433961685b1b49e7482ee5b25541815				x	x	x
CVE-2015-1471	https://github.com/delta/pragyan/commit/c93bc100ec93fc78940fbdca9b6b009101858309				x		x
CVE-2014-9573	https://github.com/mantisbt/mantisbt/commit/69c2d28d	x	x				
CVE-2014-9528	https://github.com/humhub/humhub/commit/febb89ab823d0bd6246c6cf460addabb6d7a01d4	x	x	x			
CVE-2014-9464	https://github.com/microweber/microweber/commit/4ee09f9dda35cd1b15daa351f335c2a4a0538d29	x	x	x			
CVE-2014-9096	https://github.com/Pligg/pligg-cms/commit/efb967b944375cd3ea3cd84c80d86d339dbe030e						
CVE-2014-9089	https://github.com/mantisbt/mantisbt/commit/b0021673ab23249244119bde3c7fcecd4daa4e7f		x				
CVE-2014-8351	https://github.com/LaboCNIL/CookieViz/commit/489b6050f6c53fe7b24c4bed3eeb9c25543960e2						x
CVE-2014-5017	https://github.com/LimeSurvey/LimeSurvey/commit/9938bcd1df8ea27052557c722a67b00c0e7d6cb6	x	x	x			
CVE-2014-3773	https://github.com/nilsteampassnet/TeamPass/commit/7715512f2bd5659cc69e063a1c513c19e384340f				x		x
CVE-2014-1609	https://github.com/mantisbt/mantisbt/commit/7efc01750f853e18ebfacedfd2374c4179028b3f	x	x				
CVE-2014-1608	https://github.com/mantisbt/mantisbt/commit/00b4c17088fa56594d85fe46b6c6057bb3421102		x				
CVE-2014-1401	https://github.com/auracms/AuraCMS/commit/790f66ffbc4f23a6e13636fc79d0aa1a7d81e747						
CVE-2014-10033	https://github.com/gburton/oscommerce2/commit/e4d90eccd7d9072ebe78da4c38fb048bfe31c902						
CVE-2013-4879	https://github.com/bigtreecms/BigTree-CMS/commit/c5f27bf66a7f35bd3daeb5f693f3e2493f51b1f3						x
CVE-2013-4789	https://github.com/Cotonti/Cotonti/commit/45eec046391afabb676b62b9201da0cd530360b4	x	x				x
CVE-2013-3527	https://github.com/vanillaforums/Garden/commit/83078591bc4d263e77d2a2ca2831009977552900d	x	x	x			
CVE-2013-3524	https://github.com/DavidJClark/phpVMS-PopUpNews/commit/efaffa04ef87db1722d69ac7bfc07be71ce2dccf	x	x	x			
CVE-2013-3081	https://github.com/JojoCMS/Jojo-CMS/commit/972757c4500d94b4b1306bf092e678add3a987d8	x	x	x			
CVE-2013-2559	https://github.com/symphonycms/symphony-2/commit/6c8aa4e9c810994f763283748742686 7ce50f468		x		x		x
CVE-2012-5162	https://github.com/osclass/OSClass/commit/ff7ef8a97301aaaf6a97fe46c2c27981a86b4e2f		x	x	x		
CVE-2012-3471	https://github.com/ushahidi/Ushahidi_Web/commit/3f14fa0		x	x			
CVE-2012-3470	https://github.com/ushahidi/Ushahidi_Web/commit/3301e48		x	x			
CVE-2012-3469	https://github.com/ushahidi/Ushahidi_Web/commit/e0e2b66						
CVE-2012-3468	https://github.com/ushahidi/Ushahidi_Web/commit/fdb48d1						
CVE-2012-2762	https://github.com/s9y/Serendipity/commit/87153991d06bc18fe4af05f97810487c4a340a92	x				x	x
CVE-2012-0973	https://github.com/osclass/OSClass/commit/ff7ef8a97301aaaf6a97fe46c2c27981a86b4e2f	x	x	x	x		
CVE-2011-4960	https://github.com/silverstripe/sapphire/commit/fef7c32		x				x
CVE-2011-4959	https://github.com/silverstripe/sapphire/commit/73cca09						
CVE-2011-4802	https://github.com/Dolibarr/dolibarr/commit/c539155d6ac2f5b6ea75b87a16f298c0090e535a				x		x
CVE-2011-4341	https://github.com/symphonycms/symphony-2/commit/476e4926e2773588eab10dd3036f27e1411521b5				x		x
		22	28	20	9	2	11

SIGPLAN International Workshop on Machine Learning and Programming Languages, page 35–42, 2017.

[9] PHP. https://www.php.net/manual/language.namespaces.dynamic.php, 2020.

[10] Z. P. Reynolds, A. B. Jayanth, U. Koc, A. A. Porter, R. R. Raje, and J. H. Hill. Identifying and documenting false positive patterns generated by static code analysis tools. In *2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 55–61, 2017.

[11] Samate. Juliet Test Suite. http://samate.nist.gov/SRD/testsuite.php, 2020.

[12] Samate. SARD. https://samate.nist.gov/SARD/testsuite.php, 2020.

[13] F. Schuckert. Patterns. https://github.com/fschuckert/sca_patterns, 2020.

[14] F. Schuckert, B. Katt, and H. Langweg. Difficult XSS Code Patterns for Static Code Analysis Tools. *1st Model-Driven Simulation and Training Environment for Cybersecurity*, 2019.

[15] C. J. Van Rijsbergen. *Information Retrieval, 2nd edition*. 1979.

[16] Z. Zhioua, S. Short, and Y. Roudier. Static Code Analysis for Software Security Verification: Problems and Approaches. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, pages 102–109, 2014.