Source Code Patterns of Cross Site Scripting in PHP Open Source Projects

Felix Schuckert[12], Max Hildner[1], Basel Katt[2] *, and Hanno Langweg[12]

[1] HTWG Konstanz,
Department of Computer Science,
Konstanz, Baden-Württemberg, Germany
felix.schuckert@htwg-konstanz.de
hanno.langweg@htwg-konstanz.de
maxhildner@fastmail.com

[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

To get a better understanding of Cross Site Scripting vulnerabilities, we investigated 50 randomly selected CVE reports which are related to open source projects. The vulnerable and patched source code was manually reviewed to find out what kind of source code patterns were used. Source code pattern categories were found for sources, concatenations, sinks, HTML context and fixes. Our resulting categories are compared to categories from CWE. A source code sample which might have led developers to believe that the data was already sanitized is described in detail. For the different HTML context categories, the necessary Cross Site Scripting prevention mechanisms are described.

1 Introduction

Cross Site Scripting (XSS) is on the fourth place in Common Weakness Enumeration (CWE) top 25 2011 [3] and on the seventh place in Open Wep Application Security Project (OWASP) top 10 2017 [4]. Accordingly, Cross Site Scripting is still a common issue in web security. To discover the reason why the same vulnerabilities are still occurring, we investigated the vulnerable and patched source code from open source projects. Similar methods, functions and operations are grouped together and are called source code patterns. Our work shows which source code patterns occur in real life projects, to provide a data set that can be compared to existing vulnerability data sets like SAMATE SARD [8]. The questions to be answered are: How do source code patterns from real projects look like? What main protection mechanisms are required to prevent Cross Site Scripting attacks? Real source code samples from open source projects are investigated to get answers to these questions.

This paper begins with an overview of related work in section 2. Section 3 explains how we got the source code sample and how the manual review process looks like. The next section explains what Cross Site Scripting categories from CWE [3] exist and how they fit into source code pattern categories. In section 5 our taxonomy resulting from the manual review process is explained. The sample from CVE-2012-5163 described in section 6 is a sample where developers might thought that data is already sanitized. In section 7 the results are discussed. The final section provides some discussion about how our result can be used in the future.

*Basel Katt

Proceedings of the 11th Norwegian Information Security Conference, Sep. 19-20, 2018.
NISK Journal, e-ISSN 1894-7735. https://ojs.bibsys.no/index.php/NISK

2 Related Work

For SQL injection and Buffer Overflow vulnerabilities, we created the source code patterns with the same method that was used by [23] and [22]. Research projects about general classification of source code patterns exist. Lerthathairat and Prompoon [15] did research about the classification of source code into bad code, clean code and ambiguous code. They use metrics in source code like comments, the size of the function, et cetera. These metrics were analysed by using Fuzzy logic to determine which category the source code belongs to. Bad and ambiguous code were improved by refactoring. A more security related work is from Hui et al. [14], they use a software security taxonomy for software security tests. They created a security defects taxonomy based on top 10 software security errors from authoritative vulnerabilities enumerations. Their taxonomy is categorized into *induced causes*, *modification methods* and *reverse use methods*. They advise that their taxonomy should be used as security test cases. Stivalet and Fong [24] present a tool that allows to create short code examples including software vulnerabilities. They split up the source code parts into *input*, *filtering* and *sink*. Permutations of these parts will create different examples. All of the examples can be found in the Testsuite 103 within the Software Assurance Reference Dataset [8].

The research also requires data sources, which are used to classify the source code patterns. Massacci and Nguyen [17] researched different data sources for vulnerabilities, e.g. Common Vulnerabilities and Exposures (CVE), National Vulnerability Database (NVD), et cetera. They looked into which data sources were used by other research projects. They also used Firefox as a database for their analysis. Wu et al. [25] use semantic templates created from the existing CWE database [3]. They should help to understand security vulnerabilities. The authors did an empirical study to prove that these semantic templates have a positive impact on the time to completion on finding the vulnerability.

Louw and Venkatakrishnan [16] present a tool Blueprint can be used to defend against Cross Site Scripting attacks. Different Cross Site Scripting variants are explained and how they can be exploited. Another framework from Gupta and Gupta [13] can be used to protect against XSS attacks specialized on HTML 5 web pages. Another approach from Maurya [18] [19] shows how to prevent Cross Site Scripting vulnerabilities on the server side. In their work different security levels require different sanitization approaches. Nadji et al. [20] present different XSS attack scenarios and also suggest on how to prevent these attacks using a combination of server and client protection mechanisms. Another work from Gundy et al. [12] uses a randomized approach to prevent against XSS attacks. As long as the attackers cannot predict the randomization that approach should deny any XSS exploits. In contrast we suggest using the correct standard prevention methods based on the html context to prevent XSS attacks.

3 Method

For our research we decided to review vulnerabilities that were found in open source projects. We focus on vulnerabilities that are tracked in the *CVE* [1] database. CVE reports between 2010 and 2016 (seven years) are used as data samples to ensure that developers had sufficient time to patch the vulnerabilities. To get the vulnerable and patched source code related to CVE reports the results from the source code crawler from [22] were used. It checks CVE entries for related GitHub [2] patches and downloads the vulnerable and patched source code. Table 1 shows how many source code samples for different vulnerability types and programming languages are found. We chose PHP as reviewed programming language because it provides the most samples (122) related to Cross Site Scripting. Out of these source code samples 50 CVE

Categories from CVE details	CVEs	Github	PHP	Java	C/C++	js	Python	Ruby
Denial Of Service	10,929	737	7	3	664	4	8	6
Execute Code	10,647	217	70	6	83	8	12	17
Overflow	6,594	302	3	1	280	0	5	1
Obtain Information	5,471	233	32	6	153	5	10	11
Cross Site Scripting	4,878	219	122	5	2	46	14	11
Memory corruption	3,419	52	0	0	49	0	0	0
Bypass a restriction or similar	2,609	96	24	4	51	1	6	2
Gain privileges	2,207	118	4	0	100	0	0	1
SQL Injection	1,732	71	53	3	2	2	0	5
Directory traversal	1,059	35	14	1	7	3	2	5
CSRF	1,046	33	25	2	0	6	1	3
File Inclusion	100	3	0	0	0	0	0	0
Http response splitting	74	4	0	0	0	0	0	0
Summary	50,765	2,120	354	31	1,391	75	58	62

Table 1: Software vulnerabilities in open source software grouped by vulnerability type and programming language.

reports are randomly selected for a manual review. 50 samples means 1% of all CVE entries related to XSS and 40% of all CVE entries related to XSS and PHP.

Cross Site Scripting vulnerabilities are commonly split into three parts (sources, sanitization and sinks). The sources are methods where data is provided which can be manipulated by a user. Sinks are methods where such data can be harmful. In a XSS vulnerability it will result in scripts that will be executed on the victim's browser. Sanitization methods transform user provided data such that it will not be harmful, if it reaches sinks.

The manual reviews were done as follows. For each CVE report the note entry was looked up. It usually describes which input fields or variables can be used to exploit the vulnerability. Just within the scope these variables/fields are the tainted data sources for the Cross Site Scripting vulnerabilities. If no variable/field is mentioned the source code is manually backtracked from the patched source code to find the sources. By doing a data flow analysis from the sources relevant source code patterns are tracked. For example, it will be noted if a sanitization method from a framework is used. For sources, insufficient sanitization, PHP sinks, HTML context and fixes a taxonomy is created based on the review results.

4 CWE

Common Weakness Enumeration (CWE) [3] provides categories for software vulnerabilities. CWE-79 is the basic enumeration for Cross Site Scripting. It is distinguished between reflected XSS, stored XSS and dom-based XSS. In a developer perspective these are different sources and sinks depending on when the sanitization should happen. For example, if only sanitized data should be stored in the database, the category can be seen as sink category. On the contrary it can be seen as a source category, if sanitization should only occur before the data is presented on the web page. The CWEs 81 to 87 are different variants of the base CWE-79. These can be seen as different categories for failed sanitization methods and different HTML contexts. The table 2 shows source code pattern categories of the different CWE enumerations.

CWE-81	Improper Neutralization of Script in an Error Message Web Page	HTML context
CWE-82	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page	HTML context
CWE-83	Improper Neutralization of Script in Attributes in a Web Page	HTML context
CWE-84	Improper Neutralization of Encoded URI Schemes in a Web Page	HTML context
CWE-85	Doubled Character XSS Manipulations	Failed sanitization
CWE-86	Improper Neutralization of Invalid Characters in Identifiers in Web Pages	Failed sanitization
CWE-87	Improper Neutralization of Alternate XSS Syntax	Failed sanitization

Table 2: CWE Cross Site Scripting variants mapped to categories.

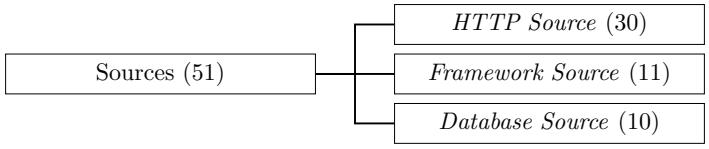

Figure 1: Taxonomy of sources based on the data set.

5 Taxonomy of Source Code Patterns

In this section the different taxonomies resulting from the review process are described in detail. The correlations to more or less corresponding CWE categories are mentioned.

5.1 Sources

Figure 1 shows three categories for the sources. There are 51 sources because CVE-2014-9270 is a stored and reflected Cross Site Scripting vulnerability. Accordingly, for each type a different source was found (database source and user input source).

HTTP Source

Sources which fall in the *HTTP Source* category are basic HTTP methods provided by PHP. No wrapper was used which might trick developers into thinking that the data might already be sanitized. Sources from this category are related to reflected Cross Site Scripting vulnerabilities. In our data set we found *$_GET*, *$_REQUEST*, *$_POST*, *$_SERVER* and *$_COOKIE*. These are reserved variables from PHP [7]. *$_FILES*, *$_HTTP_RAW_POST_DATA* would also fall into this category, but these were not found in our data set.

Framework Source

PHP is commonly used with frameworks which provide features to create web pages more conveniently. Methods that wrap around methods from the *HTTP Source* category or methods provided from frameworks to get user provided data fall into this category (e.g. *gpc_get_string()*, *getParam()*). All our samples were internally using sources from the *HTTP Source* category.

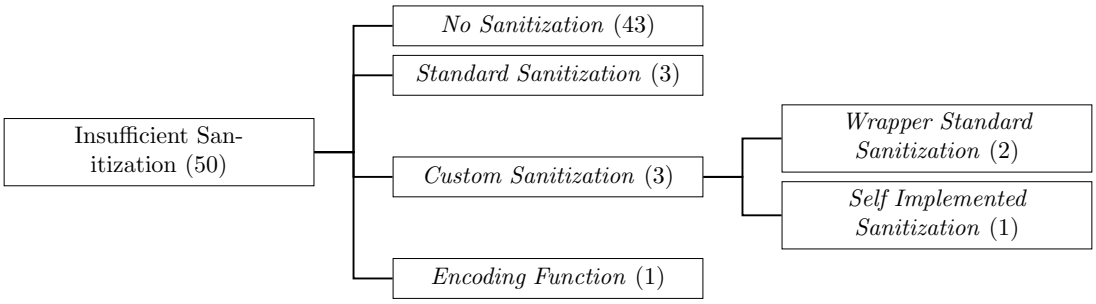

Figure 2: Taxonomy of insufficient sanitization methods based on the data set.

Accordingly, vulnerabilities with sources from this category are also related to reflected Cross Site Scripting. The extra category was created because such methods might already sanitized the input.

Database Source

Stored Cross Site Scripting stores the malicious input in the database. That input will be later on presented without any further sanitization on a web page. For our research the methods where user data is returned from the database is seen as a source. For example, in our data set we found functions like *db_fetch_row()* and *serendipity_db_query()*.

5.2 Insufficient Sanitization

There is a difference between sanitization methods and escaping methods. Sanitization methods are removing suspicious character which is commonly used to mitigate SQL injection vulnerabilities. In contrast escaping methods are just escaping suspicious character that cannot be used to inject any code. The resulting data from both methods will not be harmful anymore. Escaping methods are more commonly used to protect against Cross Site Scripting attacks because it will not remove characters that an user wants to be displayed. We group both methods together and call them sanitization methods which can either be sanitization methods or escaping methods. Most of our vulnerable samples did not use any sanitization methods. Nevertheless, some did use well known sanitization methods, but Cross Site Scripting vulnerabilities still occurred. Figure 2 shows an overview of the categories for insufficient sanitization methods.

No Sanitization

As the name already indicates, no sanitization method was used. Samples where plain user input will reach a sink fall into this category.

Standard Sanitization

Developers actually used official sanitization methods like *htmlspecialchars()*, but a Cross Site Scripting vulnerability still existed. Why was that sanitization insufficient? Two of the three found samples, the sanitization was insufficient because the HTML sink was from the category *JavaScript Context*. If sinks are from that category, the sanitization has to be more specialized because the context is already in Javascript. How to prevent XSS attacks in such a context is

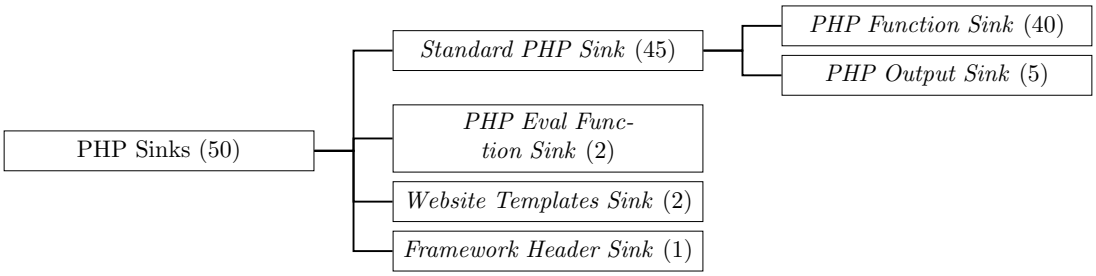

Figure 3: Taxonomy of PHP sinks based on the data set.

described later on. The last sample did have a special condition where the sanitization method is not used. That sample is described in detail in section 6.

Custom Sanitization

Custom sanitization methods were also found in the data set. Two samples did use wrapper methods which internally use methods from the *Standard Sanitization* category. Accordingly, they should be protected against XSS attacks, but the sinks are again from the *JavaScript Context* category.

One sample did implement a sanitization method using *str_replace()* method. The implementation was insufficient and the patch did fix the vulnerability by using a sanitization method from the category *Standard Sanitization*.

Encoding Function

One sample did use an encoding function. These functions are not supposed to be used as a sanitization method. Nevertheless, as seen in the fixes categories, some developers patched the vulnerabilities by using encoding functions. This just prevents Cross Site Scripting vulnerability as long the context is from the *Plain HTML Context* category and the correct HTML encoding function is used.

5.3 PHP Sinks

The sink categories are split into PHP and HTML sinks. An overview of the PHP sink categories is shown in figure 3. This sinks taxonomy is useful for developers because these are the sinks where user input without sanitization will be harmful.

Standard PHP Sink

This category covers standard output to the web page. Most of our data samples did have sinks in the *Standard PHP Sink* category. It is split into the first category *PHP Function Sink* where methods are used to output the data. Common methods are *echo()* and *print()*. The other category *PHP Output Sink* covers simple output in PHP files, where the *<?...?>* element is used.

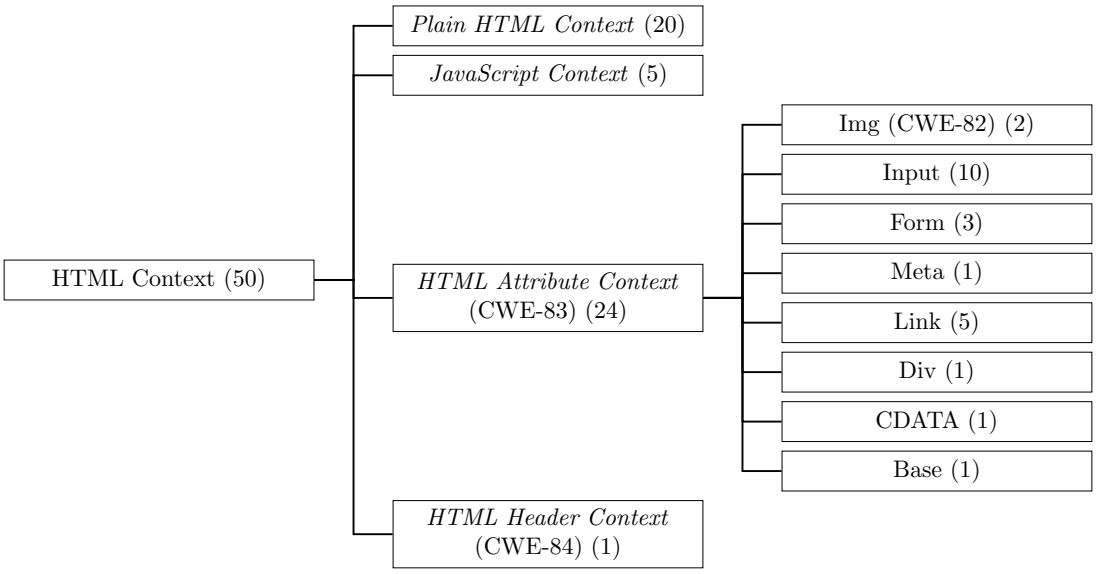

Figure 4: Taxonomy of HTML sinks based on the data set.

PHP Eval Function Sink

In our data set two samples did have an *eval()* function as sink. These CVE reports were marked as Cross Site Scripting vulnerability. Actually these sinks also open up Direct Dynamic Code Evaluation vulnerabilities (CWE-95). Nevertheless, it can also result in a Cross Site Scripting vulnerability.

Website Templates Sink

Some PHP frameworks provide template files, which can be used to write PHP similar code with some extra features. In our data set two samples are using *tlp* files from the Smarty framework [9]. If such a framework template is used it falls into this category.

5.3.1 *Framework Header Sink*

One sample did have the sink in a header parameter. The PHP framework used in that sample provides a function to set a header parameter. Accordingly, this category was created for sinks which allow to modify the HTTP header.

5.4 HTML Context

The PHP sinks categories are more focused on what functions are used to print the data. HTML context rather focus on where the data is presented in the web page. This taxonomy is more similar to the categories provided by CWE. Figure 4 shows the categories found in our data set.

HTML Context	Encoding	Sanitization	Sanitization & Escaping
Plain HTML Context	prevent	prevent	prevent
HTML Attribute Context	insufficient	insufficient	prevent
HTML Header Context	insufficient	prevent	prevent
JavaScript Context	insufficient	insufficient	prevent

Table 3: CWE Cross Site Scripting variants mapped to categories.

Plain HTML Context

Outputs which get into a context of this category are in a simple plain HTML part. No special condition like being in a Javascript context or being an attribute. Standard sanitization methods are well suited for such a context. Also the use of HTML encoding functions is enough as long the output is inside the HTML body [6].

JavaScript Context

Sinks where the output will be in a Javascript context fall into this category. Accordingly, sanitization must be more specialized. The OWASP XSS prevention sheet [6] explains that simple encoding functions are not enough. The output also should be quoted and sanitized to prevent any Cross Site Scripting attacks [6]. It is important to know that inputs must be data only. Otherwise, the prevention of Cross Site Scripting will be very difficult and requires further restrictions. In our data set only data was used inside a script tag. Another pitfall in this context is the method *htmlspecialchars()* because it does not remove simple quotes without setting the *ENT_QUOTES* parameter. If developers use simple quotes as escaping and do not set the parameter, XSS attacks are still possible.

HTML Attribute Context

This is the same category as CWE-83. The output is inside a HTML attribute. The CWE-82 is specialized version of being an image attribute. In our data set only two samples actually were in a image attribute context. The input has to be sanitized and quoted like in the *JavaScript Context* category.

HTML Header Context

One sample did have a sink from the category *Framework Header Sink*. Accordingly, the context is a HTTP header and that sample falls into this context category. To prevent any Cross Site Scripting attacks in a header, sanitization methods from *Standard Sanitization* are required. Encoding function will not be sufficient.

5.4.1 Sanitization in different contexts

As already mentioned, different HTML context require different sanitization methods. The table 3 provides an overview of what sanitization methods are sufficient enough to prevent any Cross Site Scripting attacks. Accordingly, it would be helpful to sanitize and escape any user input to be protected against XSS attacks in all HTML contexts.

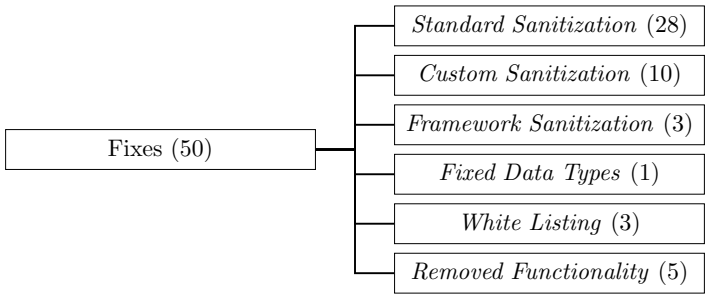

Figure 5: Taxonomy of HTML sinks based on the data set.

5.5 Fixes

An important part to prevent Cross Site Scripting is correct sanitization of user input. Most reviewed projects had no sanitization before the patch related to the CVE report was developed. The patches contain different fixes to sanitize inputs. Figure 5 shows the taxonomy created for the fixes which were used in our data set.

Standard Sanitization

Fixes of this category used the standard functions provided by PHP. No combination of multiple sanitization methods or a sanitization method from a framework is used.

In our sample, common methods were *htmlentities()*, *htmlspecialchars()*, *urlencode()*, etc.

Custom Sanitization

Some patches did fix the vulnerability by using a custom developed sanitization method. Few used a combination of different sanitization methods from the category *Standard Sanitization* as sanitization method. Also some samples did fix the vulnerability by using a sanitization method which did use replace methods like *preg_replace()*.

Framework Sanitization

Three samples were using sanitization methods provided by a framework to fix the vulnerability. Fixes that use a framework sanitization method falls into this category. For example, the *esc_html()* function from WordPress [10] fall into this category.

Fixed Data Types

One sample did use a fixed data type to prevent any Cross Site Scripting vulnerabilities. An ID value was evaluated as an integer value. Accordingly, a simple and elegant way of fixing the vulnerability.

White Listing

White listing is a common way to prevent any injection attacks. Just fixed values are allowed and all other inputs will be ignored. Three of the samples used white listing to fix the vulnerabilities.

Removed Functionality

Another category to fix a vulnerability is to remove that output. Even some sample did remove some functionality which contained the vulnerability. At least it fixed the vulnerability.

6 Special case: CVE-2012-5163

To get a better understanding of the manual review process, this section provides an example of the CVE-2012-5163 report in the open source project Osclass [5]. Looking into the report notes reveals that the id parameter in *enable_category* can be used to inject arbitrary code. Accordingly, the source is already known. Looking into the vulnerable source code reveals that the function *Params::getParam("id")* is used which falls in the ***Framework Source*** source category. The code snippet 6 shows the related source code. As seen on line 9 and 14 standard sanitization methods are used. Consequently, it falls in the insufficient sanitization category ***Standard Sanitization***. Nevertheless, because *$htmlencode* is on default set to false, the sanitization method is not used. The function will return the value without any sanitization for the *enable_category* id. As this sample shows, a developer might think that the *getParam()* function does already sanitize the inputs but in specific conditions it does not.

The code snippet 7 shows a shortened version of the *doModel()* method, which prints the not sanitized user input. Line 4 and 6 shows a little part of the data flow that was tracked by the manual review. The final PHP sink is found on line 7, where the output will be encoded by the function *json_encode* and the sink is the *echo* function. Accordingly, as PHP sink was found from the ***Standard PHP Sink*** category. The *doModel()* functions creates a web page where the found echo result in a plain HTML context. Therefore it falls into the ***Plain HTML Context*** category. The fix was very simple by encapsulating the related *getParam()* call with a *strip_tags()* functions. Thus the fix category is ***Standard Sanitization*** because it is a function provided by PHP. The encoding function is supposed to be used for a JSON context. Accordingly, this does not prevent any XSS attacks in a HTML context.

7 Discussion

The results did show that Cross Site Scripting vulnerabilities have a high rate of vulnerabilities where no sanitization is used. The vulnerable source code for CVE-2013-0807, CVE-2014-8793 and CVE-2013-4880 reports have the vulnerabilities including source and sink in one line of code. These results show that Cross Site Scripting is not as present in developers' minds as it should be. For Cross Site Scripting the different HTML contexts are relevant because as described in section 5.4.1 they require different sanitization methods. Accordingly, the prevention mechanisms and context categories have to fit to prevent any attacks.

The CWE sub categories for Cross Site Scripting are not very detailed. As our results show many different categories exist. Especially the context category *JavaScript Context* should exist because it requires more specialized sanitization to prevent any XSS attacks. The method *htmlspecialchars()* should probably also escape simple quotes as default because developers might not read the documentation carefully enough to know that an additional parameter is required to escape simple quotes. In a *JavaScript Context*context it opens up unnecessary XSS vulnerabilities.

```
static function getParam($param, $htmlencode = false)
{
    if ($param == "") return '' ;
    if (!isset($_REQUEST[$param])) return '' ;

    $value = $_REQUEST[$param];
    if (!is_array($value)) {
        if ($htmlencode) {
            return htmlspecialchars(stripslashes($value), ENT_QUOTES);
        }
    }

    if(get_magic_quotes_gpc()) {
        $value = strip_slashes_extended($value);
    }

    return ($value);
}
```

Figure 6: Params::getParams() function with insufficient sanitization from CVE-2012-5163.

```
// root category
if( $aCategory['fk_i_parent_id'] == '' ) {
...
$aUpdated[] = array('id' => $id) ;
...
$result['affectedIds'] = $aUpdated ;
echo json_encode($result) ;
break ;
}
...
break ;
```

Figure 7: A shortened version of the sink source code part from CVE-2012-5163.

8 Conclusion and Future Work

We analysed the source code of 50 GitHub projects which are correlated to CVE reports mentioning Cross Site Scripting and available source code patches. The results show different taxonomies for important source code patterns. Relations to the existing CWE categories are created. Our taxonomy is more focused on the developers point of view. In combination with our previous work [22], [23], three taxonomies for different vulnerabilities categories are created. These taxonomies allow further research using the taxonomy and the source code samples as a dataset. These categories can be used to get a better understanding where Cross Site Scripting vulnerabilities can occur. Especially, the HTML context taxonomy has a big influence on what prevention mechanisms should be used.

The sample set was very small; more samples should be analysed for the source code patterns and compared to our results. Another interesting aspect would be to research source code patterns of XSS samples in the programming language JavaScript. These patterns could also

be compared to our results. This taxonomy could be used to improve the teaching of software security skills for developers. The knowledge of source code patterns can be used to create exercises. An interesting point will be to create these exercises automatically similar to previous research projects [21] [11]. Existing source code from projects can be used and transformed to create these source code patterns. Another interesting point will be using these categories to test static code analysis tools. These can be investigated whether they detect all permutations. It will be interesting to see what combinations of the categories are difficult to be detected from static code analysis tools.

References

[1] CVE Common Vulnerabilities and Exposures. https://cve.mitre.org/.

[2] GitHub. https://github.com/.

[3] CWE - Common Weakness Enumeration. http://cwe.mitre.org/, 2015.

[4] OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2015.

[5] Osclass - open source classified. https://osclass.org/, 2018.

[6] XSS (Cross Site Scripting) Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, 2018.

[7] PHP - reserved variables. http://php.net/manual/en/reserved.variables.php, 2018.

[8] SAMATE - SARD, 2018. URL https://samate.nist.gov/SARD.

[9] Smarty PHP Template Engine. https://www.smarty.net/, 2018.

[10] WordPress. https://wordpress.org/, 2018.

[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pages 110–121, 2016. doi: 10.1109/SP.2016.15.

[12] M. V. Gundy, M. V. Gundy, H. Chen, and H. Chen. Noncespaces: using randomization to enforce information ow tracking and thwart cross-site scripting attacks. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 1–18, 2009.

[13] S. Gupta and B. B. Gupta. Js-san: defense mechanism for html5-based web applications against javascript code injection vulnerabilities. *Security and Communication Networks*, 9 (11):1477–1495, 2016.

[14] Z. Hui, S. Huang, B. Hu, and Z. Ren. A taxonomy of software security defects for SST. *Proceedings - 2010 International Conference on Intelligent Computing and Integrated Systems, ICISS2010*, pages 99–103, 2010. doi: 10.1109/ICISS.2010.5656736.

[15] P. Lerthathairat and N. Prompoon. An approach for source code classification to enhance maintainability. *Proceedings of the 2011 8th International Joint Conference on Computer Science and Software Engineering, JCSSE 2011*, pages 319–324, 2011. doi: 10.1109/JCSSE.2011.5930141.

[16] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. pages 331–346, May 2009. ISSN 1081-6011. doi: 10.1109/SP.2009.33.

[17] F. Massacci and V. H. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 4:1—-4:8, 2010. doi: http://doi.acm.org/10.1145/1853919.1853925.

[18] S. Maurya. Positive security model based server-side solution for prevention of cross-site scripting attacks. *12th IEEE International Conference Electronics, Energy, Environment, Communication, Computer, Control: (E3-C3), INDICON 2015*, pages 1–5, 2016. doi: 10.1109/INDICON.2015.7443473.

[19] S. Maurya and A. Singhrova. Cross Site Scripting Vulnerabilities and Defences: A Review. *International Journal of Computer Techonology and Applications*, 6(June):478 – 482, 2015.

[20] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. *Proceedings of the Network and Distributed System Security Symposium*, pages 1–42, 2009.

[21] F. Schuckert. PT: Generating Security Vulnerabilities in Source Code. In M. Meier, D. Reinhardt, and S. Wendzel, editors, *Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit*, pages 177–182, Bonn, 2016. Gesellschaft für Informatik e.V.

[22] F. Schuckert, B. Katt, and H. Langweg. Source Code Patterns of SQL Injection Vulnerabilities. *International Conference on Availability, Reliability and Security*, 2017. doi: 10.1145/3098954.3103173.

[23] F. Schuckert, M. Hildner, B. Katt, and H. Langweg. Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox. *Proceedings of Sicherheit 2018*, pages 107–118, 2018. doi: 10.18420/sicherheit2018_08.

[24] B. Stivalet and E. Fong. Large Scale Generation of Complex and Faulty PHP Test Cases. *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pages 409–415, 2016.

[25] Y. Wu, H. Siy, and R. Gandhi. Empirical results on the study of software vulnerabilities. *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 964–967, 2011.