Experience Report: An Empirical Study of PHP Security Mechanism Usage

Johannes Dahse and Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

The World Wide Web mainly consists of web applications written in weakly typed scripting languages, with PHP being the most popular language in practice. Empirical evidence based on the analysis of vulnerabilities suggests that security is often added as an ad-hoc solution, rather than planning a web application with security in mind during the design phase. Although some best-practice guidelines emerged, no comprehensive security standards are available for developers. Thus, developers often apply their own favorite security mechanisms for data sanitization or validation to prohibit malicious input to a web application.

In the context of our development of a new static code analysis tool for vulnerability detection, we studied commonly used input sanitization or validation mechanisms in 25 popular PHP applications. Our analysis of 2.5 million lines of code and over 26 thousand *secured* data flows provides a comprehensive overview of how developers utilize security mechanisms in practice regarding different markup contexts. In this paper, we discuss these security mechanisms in detail and reveal common pitfalls. For example, we found certain markup contexts and security mechanisms more frequently vulnerable than others. Our empirical study helps researchers, web developers, and tool developers to focus on error-prone markup contexts and security mechanisms in order to detect and mitigate vulnerabilities.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, product metrics*

General Terms

Security, Measurement

Keywords

Static analysis, input sanitization, input validation, PHP

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA

1. INTRODUCTION

Empirical studies of security vulnerabilities found in the last years [3, 7, 28, 31] indicate that web-related vulnerabilities such as cross-site scripting (XSS) and SQL injection (SQLi) are among the most common software flaws. In fact, such vulnerabilities are more frequently detected in the recent years than memory corruption vulnerabilities. This observation suggests that security is only added as an ad-hoc solution in web applications, rather than planning such applications with security in mind during the design phase.

Although some best-practice guidelines on secure web programming emerged (e.g., recommendations by OWASP [17]), no comprehensive security standards are available for developers. This leads to the observation that each developer applies his own favorite security mechanisms for data sanitization or validation. As a result, many programming patterns emerged for input sanitization (e.g., type casting, data encoding, converting data to HTML entities, or prepared statements) and input validation (e.g., type validation, format validation, or whitelisting). Each of these patterns has its own advantages and drawbacks, and programming mistakes due to common pitfalls can still lead to vulnerabilities. As a side effect, this variety in security mechanisms leads to many false reports in static code analysis tools, which have to detect and analyze these security mechanisms precisely.

PHP is the most popular server-side scripting language on the Web, running on 82% of all websites [30]. It is a weakly typed language, meaning that a memory location can hold a value of every possible data type. In contrast to a strongly typed language, the weak typing in PHP implies that more security checks have to be implemented by the developer to ensure a safe usage in scenarios where only a specific data type is allowed. Furthermore, weak typing introduces oddities when comparing data of different data types, leading to even more (potential) bugs or insecurities [18].

Based on our experience in the development of a static code analysis tool for finding security vulnerabilities in modern PHP applications [4–6], we collected typical programming patterns used by developers to sanitize and validate data. In this paper, we provide a comprehensive empirical analysis of the 20 different data sanitization and validation mechanisms we found being used in web applications. Furthermore, we also study common implementation pitfalls that lead to vulnerabilities in practice. We extended our prototype in order to enumerate the security mechanisms used in PHP applications and applied it to 25 of the most popular web applications. We analyzed more than 2.5 million lines of code and present an analysis of the detected security

patterns in more than 26 thousand data flows. Our analysis helps to answer the following essential questions for PHP developers, code auditors, and static analysis engineers to refine the focus during vulnerability detection:

- **Q1**: Which security mechanisms are available?
- **Q2**: Which pitfalls might these mechanisms imply?
- **Q3**: Which security mechanisms are used how often in modern (web) applications?
- **Q4**: Which security mechanism is used to prevent which vulnerability type in which markup context?
- **Q5**: Which pitfalls occur in practice?

We believe that our work is of great value for designers of static code analysis tools, especially since the basic insights can also be applied to programs implemented in other languages. Developers can understand how to use the correct sanitization and validation mechanism in which situation.

In summary, we make the following three contributions:

- We survey common mechanisms to sanitize and validate data in PHP applications and highlight the pitfalls that can occur to answer questions **Q1** and **Q2**.
- We present our approach to enumerate these mechanisms via static code analysis techniques.
- We evaluate our approach with 25 popular real-world applications. To the best of our knowledge, this is the largest study on the usage of security mechanisms in modern PHP applications. Based on this analysis, we can answer research questions **Q3** – **Q5**.

2. TYPES OF SECURITY MECHANISMS

Web applications receive remotely supplied (and potentially malicious) user data via HTTP request parameters. In PHP applications, *user input* is always received as data of type *string*. If this input is processed in a security sensitive operation of the web application, a vulnerability can occur. For example, a dynamically built SQL query with embedded user input may lead to a SQL injection vulnerability [8]. In order to prevent such a critical vulnerability, the user input has to be sanitized or validated beforehand. For this purpose, a *security mechanism* is applied between the user input (*source*) and the sensitive operation (*sink*) such that malicious data cannot reach the sensitive operation.

To study the variety of security mechanisms used by developers in practice (**Q1**), we collected common types of mechanisms we experienced in real-world applications during the development of our static analysis tool [4–6]. Although the possible ways of implementation are endless, our list covers all different mechanisms we encountered. They can be grouped into *input sanitization* (see Section 2.1) and *input validation* (see Section 2.3) mechanisms. Some sanitization and validation mechanisms have to be applied carefully to the context of the markup (see Section 2.2 and Section 2.4). Additionally, a security mechanism can be applied *path sensitively* (see Section 2.5). Each mechanism is illustrated with an example and common pitfalls are explained (**Q2**).

2.1 Generic Input Sanitization

Generally speaking, during input *sanitization* the data is transformed such that harmful characters are removed or defused. The advantage of this approach is that relevant parts of the data stay intact, while only certain characters are removed or replaced. This way, the application can proceed with the sanitized data without a request for resubmission. In the following, we present several ways to sanitize data *generically* against all type of injection flaws, such as cross-site scripting [13] and SQL injection [8].

2.1.1 Explicit Type Casting

Numeric characters can be safely used in security sensitive string operations. To ensure only numeric characters, a string can be explicitly typecasted to a number by using the *typecast* operator or built-in functions.

```
1 $var = (int)$var;        // safe
2 $var = intval($var);     // safe
3 settype($var, 'int');    // safe
```

Listing 1: Examples for explicit type casting.

All three operations in Listing 1 ensure a secure use of the variable `$var` regarding injection flaws. PHP uses *duck typing* to determine the integer value of a string. An empty and non-empty string is typecasted to the number 0. However, if the string starts with a number (*"123abc"*), the number is used as result of the typecast (*123*). We will introduce pitfalls associated with duck typing later on.

2.1.2 Implicit Type Casting

Similar to an explicit typecast, an implicit type cast automatically occurs if data is used in mathematical operations. Listing 2 shows an addition in line 1 in which `$var` is safely typecasted to integer before a number is added.

```
1 $var = $var + 1;   // safe
2 $var = $var++;     // unsafe
```

Listing 2: Examples for implicit type casting.

In contrast, the *increment* operator in line 2 performs no typecast and also works on strings. For example, the last character in the string *aaa* will be incremented to *aab*. Thus, `$var` can still contain malicious characters.

2.1.3 Formatting

Type casting is also performed by PHP's built-in *format string* functions. Different *specifiers* (beginning with a percentage sign) can be used in a format string that determine the data type of the data they will be replaced with. An example is given in Listing 3 that uses the identifier `%s` (string) and `%d` (numeric).

```
1 $var = sprintf("%s %d", $var1, $var2);   // unsafe / safe
```

Listing 3: Sanitization with a format string function.

The argument `$var1` is unsafely embedded to the string assigned to `$var`. Contrarily, `$var2` is safely typecasted to integer before it is embedded to the string.

2.1.4 Encoding

Exploitation of injection flaws almost always requires special characters. Thus, next to numbers, alphabetical letters can be considered to be a safe character set. By encoding data to an alphanumeric character set, the data is sanitized. Listing 4 provides a few encoding examples.

Although the *base64* and *url* encoding introduces a few special characters (`+`, `/`, `=`, or `%`), they are generally not sufficient to form a malicious payload and these encodings can be

considered as safe when used in a sensitive sink. Other encodings, however, include the full set of ASCII characters in the transformed output and thus are unsafe to use in sinks. Specifically, the transformation or *decoding* to the original data is unsafe because it reanimates malicious characters.

```
1 $var = base64_encode($var);      // safe
2 $var = urlencode($var);          // safe
3 $var = zlib_encode($var, 15);    // unsafe
4 $var = urldecode($var);          // unsafe
```

Listing 4: Transforming data into different encodings.

2.1.5 *Filtering*

It is also possible to sanitize data by built-in filter functions. If the data passes a filter, it is returned unmodified. Otherwise, *false* is returned so that the function can also be used for input validation (see Section 2.3). Listing 5 demonstrates the usage of two filter functions.

```
1 $var = filter_var($var, FILTER_VALIDATE_INT);    // safe
2 $var = filter_var($var, FILTER_VALIDATE_EMAIL);  // unsafe
3 $vars = array_filter($vars, 'is_numeric');       // safe
4 $vars = array_filter($vars, 'is_file');          // unsafe
```

Listing 5: Sanitization with a filter.

While the filter for integer/numeric values is safe, filtering for valid email addresses or files is not necessarily, because the character set of email addresses and file names allow special characters. For example the SQL injection payload `1'or'1'-@abc.com` can be a valid email and file name.

2.2 Context-Sensitive Input Sanitization

In contrast to generic input sanitization, context-sensitive sanitization removes or transforms only a small set of special characters to prevent exploitation of a specific vulnerability type or a subset of vulnerabilities. Therefore, sanitized data may still cause a vulnerability when used in the wrong markup context or another type of sensitive sink. Again, we provide in the following examples for security mechanisms and common pitfalls inspired by real-world code we found.

2.2.1 *Converting*

A common method to distinguish between HTML markup characters and data is to convert markup characters within data to HTML entities. In Listing 6, the built-in function `htmlentities()` is applied to different HTML contexts.

```
1 $var = htmlentities($var);
2 echo '<a href="abc.php">'.$var.'</a>';            // safe
3 echo '<a href="abc.php?var='.$var.'">link</a>';   // safe
4 echo "<a href='abc.php?var=".$var."'>link</a>";   // unsafe
5 echo "<a href=abc.php?var=".$var.">link</a>";     // unsafe
6 echo '<a href="'.$var.'">link</a>';               // unsafe
```

Listing 6: Converting meta characters to HTML entities.

The function `htmlentities()` converts the < and > character to the entity `<` and `>`, as well as the double-quote character to `&quot;`. Thus, the data is safely used in line 2, where no new HTML tag can be opened with a < character, and in line 3, where no double-quote can be used to break the *href* attribute. However, if single-quotes (line 4) or no quotes (line 5) are used for the attribute, an attacker can inject eventhandlers to execute JavaScript code. In line 6, double-quotes are used and cannot be broken, but a *javascript:* protocol handler can be injected at the beginning of the URL attribute and craft a malicious link.

2.2.2 *Escaping*

In SQL markup, string values are *escaped* in order to prevent breaking the quotes the value is embedded in. A prefixed backslash before a quote tells the SQL parser to interpret the next quote as data instead of syntax.

```
1 $var = addslashes($var);
2 $sql = "SELECT * FROM user WHERE nr = '".$var."'"; // safe
3 $sql = 'SELECT * FROM user WHERE nr = "'.$var.'"'; // safe
4 $sql = "SELECT * FROM user WHERE nr = ".$var;      // unsafe
5 mysql_query($sql);
```

Listing 7: Escaping data for a SQL query.

In Listing 7, a value is escaped with the built-in function `addslashes()`. It prevents breaking a single- or double-quoted string value (line 2 and line 3). However, when no quotes are used in the SQL query (line 4), breaking quotes is irrelevant and an attacker can inject SQL syntax.

Furthermore, truncating a string *after* it was escaped introduces a security risks. If the string is truncated at an escaped character, a backslash remains unescaped at the end of the string that breaks any upcoming quote in the query.

2.2.3 *Preparing*

A safer way to separate data and SQL syntax is to use *prepared statements* (see Listing 8). Here, the SQL statement is prepared with place holders for parameters. Data can then be bound to each place holder which will be safely inserted at runtime, regardless of quoting or data type.

```
1 $stmt = $db->prepare("INSERT INTO ".$pfx."user (id, name)
                        VALUES (?, ?)");
2 $stmt->bind_param("i", $var); // safe
3 $stmt->bind_param("s", $var); // safe
4 $stmt->execute();
```

Listing 8: Binding parameters to a prepared statement.

Note that if the SQL statement is prepared dynamically, it is still vulnerable to SQL injection. In line 1, the table prefix variable `$pfx` can still inject SQL syntax. Another pitfall to be aware of is that the inserted *name* to the table *user* can still cause a *second-order* vulnerability [5].

2.2.4 *Replacing*

Manual replacing of certain characters is error-prone in practice. In Listing 9, two ways of replacing single-quotes are shown that look safe at first sight.

```
1 $var = str_replace("'", "", $var);    // unsafe
2 $var = str_replace("'", "\'", $var);  // unsafe
3 $sql = "INSERT INTO user VALUES ('".$var."','".$var."')";
4 mysql_query($sql);
```

Listing 9: Two examples for manual escaping.

In line 1, single quotes are removed completely and in line 2 they are escaped with a backslash. However, the backslash itself is forgotten in both replacements. Hence, a backslash can be injected to break the single quotes. The second replacement will replace "\'" to "\\'", which escapes the backslash and leaves the single quote unescaped.

2.2.5 *Regex Replacing*

Regular expressions (*regex*) can be used for string replacement and are error-prone if not specified carefully [2]. For example, in Listing 10, all characters except for those specified in brackets shall be removed to ensure safe data output.

```
1 $var = preg_replace("/[^a-z0-9]/", "", $var); // safe
2 $var = preg_replace("/[^a-z.-_]/", "", $var); // unsafe
3 echo $var;
```

Listing 10: String replacement with regular expressions.

The first regular expression allows alphanumerical characters. The second regular expression could intent to allow lowercase letters as well as the *dot*, *minus*, and *underscore* character. However, the full ASCII range between the *dot* and *underscore* character is allowed, including the character < and > that allow to inject HTML.

2.3 Generic Input Validation

Next to input *sanitization* that transforms data into a safe character set, data can be simply refused if it does not hold a condition or fails a check. This input *validation* ensures that only data which already consists of a safe character set reaches a sensitive sink and data containing malicious characters is refused. In the following, we introduce generic conditions and checks to validate data against all type of injection flaws we empirically found during our analysis.

2.3.1 Null Validation

The easiest way to validate that no malicious character is within a given string is to check if it is empty or not set (see Listing 11). However, this also implies that no data can be used. A null validation is commonly used in combination with a previous `unset()` operation. A static code analysis tool should be able to calculate the boolean logic behind a `not` operator and multiple `else` or `elseif` branches (line 4).

```
1 if(empty($var)) { }           // safe
2 if(!isset($var)) { }          // safe
3 if(!$var) { }                 // safe
4 if(empty($var)) { } else { }  // unsafe
```

Listing 11: Validating a variable's initialization.

2.3.2 Type Validation

Validation can also be performed by checking the data type. Listing 12 shows four examples that check for a numeric data type. In line 3, PHP's *duck typing* is used when a string is provided for an integer typecast. According to its rules, the typecast result of a string that starts with a number will bypass the validation. The same applies to the validation in line 4, however, `$var` is sanitized by overwriting it with the typecast result.

```
1 if(is_numeric($var)) { }        // safe
2 if(is_int($var) === true) { }   // safe
3 if((int)$var) { }               // unsafe
4 if($var = (int)$var) { }        // safe
```

Listing 12: Validating a variable's type.

2.3.3 Format Validation

Next to the data type, a specific data format can be enforced. For example, the time and date format ensures that no malicious payload can be crafted with the given set of characters (see Listing 13). Other formats, however, might allow malicious characters, such as parts of the URL format.

```
1 if(checkdate($var)) { }           // safe
2 if($var = strtotime($var)) { }    // safe
3 if($vars = parse_url($var)) { }   // unsafe
```

Listing 13: Validating a variable's format.

2.3.4 Comparing

By comparing input against a specific non-malicious value, the data is implicitly limited to this value. In PHP, this can be done by the *equal* operator, the *identical* operator, or built-in functions (see Listing 14).

```
1 if($var == 'abc') { }          // safe
2 if($var === 'abc') { }         // safe
3 if(!strcmp($var, 'abc')) { }   // safe
4 if($var == 1) { }              // unsafe
5 if($var === 1) { }             // safe
```

Listing 14: Validating a variable's string content.

Care should be taken when using the *equal* operator (`==`, line 4). It performs a type unsafe comparison by using duck typing on operands. Therefore, any string starting with the number *1* is typecasted to the integer *1* when compared with an integer. Thus, malicious characters in this string bypass the comparison to *1*. A type safe comparison is performed with the *identical* operator (`===`).

2.3.5 Explicit Whitelisting

To compare input against a set of whitelisted values, an array can be used as lookup table, as shown in Listing 15. The lookup can be performed either by array key (line 2 to line 4) or array value.

```
1 $whitelist = array('a' => true,'b' => true,'c' => true);
2 if(isset($whitelist[$var])) { }              // safe
3 if($whitelist[$var]) { }                     // safe
4 if(array_key_exists($var, $whitelist)) { }   // safe
5 if(in_array($var, array('a','b','c'))) { }   // safe
6 if(in_array($var, array(1, 2, 3))) { }       // unsafe
7 if(in_array($var, array(1, 2, 3), true)) { } // safe
```

Listing 15: Using an explicit whitelist for validation.

Looking up a value in an array applies to the same rules than comparing two values with the *equal* operator. Thus, the example in line 6 is unsafe because the string *1abc* is typecasted to *1* and found successfully in the array. To avoid this, the *strict* parameter has to be set to *true*. Similar pitfalls occur when using the built-in function `array_search()`.

2.3.6 Implicit Whitelisting

Next to an array, a value can be compared against a fixed set of items. For example, method and property names are limited to an alphanumerical character set . If a value matches one of these (`method_exists()`), it implies that no malicious character is contained.

2.3.7 Second-order Validation

Similar to a whitelist, a value can be looked up in a resource, such as the file system or a database. Listing 16 shows an example where an email is looked up in the table *user*. Only if a user with the email address exists, the path is reached. Similarly, three additional examples show a check for the presence of a file name.

```
1 $var = addslashes($var);
2 $r = mysql_query("SELECT * FROM user WHERE mail='$var'");
3 if(mysql_num_rows($r)) { }
4 if(file_exists($var)) { }
5 if(realpath($var)) { }
6 if(stat($var)) { }
```

Listing 16: Database and file name lookup.

The safety of the validation depends on the present values in the database or available file names. If the application allows to insert arbitrary email addresses to the database or to upload arbitrary file names, the validation is unsafe [5].

2.4 Context-Sensitive Input Validation

Input validation can also be performed context-sensitively: for a subset of vulnerability types, the data is validated against a safe character set or the absence of malicious characters regarding the vulnerability type and markup context in a specific code path. Another vulnerability type or another markup context within the same path may still be exploitable. In the following, we introduce examples for context-sensitive input validation we found in our study.

2.4.1 Searching

For a specific context, user input can be validated by proofing the absence of a malicious character required for exploitation. For example, if no < character is found in the input, it can be considered as safe regarding XSS in the context of a HTML tag. Two typical search examples are shown in Listing 17.

```
1 if(!strpos($var, '<')) { }            // unsafe
2 if(strpos($var, '<') === FALSE) { }   // safe
```

Listing 17: Searching for a specific malicious character.

The first example is unsafe, because `strpos()` returns the offset at which the character was found in the string. If the string starts with a < character, offset *0* is returned that evaluates to *false* in the `if`-condition. Thus, the first validation can be bypassed.

2.4.2 Length Validation

Note that a specific string length can or cannot prevent exploitation, depending on the vulnerability type and its markup context. For example, on MySQL, the SQL injection of the three characters `'-'` is equal to a `'or'1'='1` injection. For a XSS vulnerability, three characters are usually not enough for exploitation. Thus, a string length validation, as shown in Listing 18, is context-sensitive.

```
1 if(strlen($var) < 3) { }
```

Listing 18: Validating the length of a variable.

2.4.3 Regular Expressions

Regular expressions are a useful tool to perform very precise input validation. In Listing 19, three different examples are shown to allow only alphanumerical characters in the following path.

```
1 if(!preg_match('/[^\w]/', $var)) { }   // safe
2 if(preg_match('/\w+/', $var)) { }      // unsafe
3 if(preg_match('/^\w+$/', $var)) { }    // safe
```

Listing 19: Validating the character set with regex.

The first example ensures, that no characters are present except for alphanumerical (`\w`) characters. The second example checks that alphanumerical characters are present. However, it fails to check the complete string range due to the missing boundary checks (compare to line 3). Hence, one alphanumerical character at any position of the string is enough to bypass the validation. More pitfalls regarding regular expressions can be found in Section 2.2.5.

2.5 Path Sensitivity

A security mechanism can also be spread across multiple paths of the control flow. In this case, path-insensitive code analysis reports false positives when impossible path combinations are considered [38]. In the following, we present examples for path-sensitive applications of security mechanisms and outline the challenges for static code analysis.

2.5.1 Path-sensitive Sanitization

In Listing 20, the variable `$var` is implicitly sanitized by first checking for a numerical data type. If this condition does not hold, the variable is sanitized. For example, in line 2, the variable is set to the integer *0* which effectively limits the variable to numerical characters for all paths after the `if`-block. Similarly, the variable could be unset (line 3) or context-sensitive sanitization could be applied (line 4).

```
1 if(!is_numeric($var)) {
2   $var = 0;
3   //unset($var);
4   //$var = addslashes($var);
5 }
```

Listing 20: Path-sensitive sanitization.

Typically, static code analysis tools fail to recognize this type of input sanitization because all execution paths are considered separately. Thus, it is assumed that the variable is not modified when the `if`-path is not taken. However, this implies that the variable's value is already numerical.

2.5.2 Path-sensitive Termination

A similar confusion of static analysis can occur when the program is terminated based on input validation. In Listing 21, the program execution is halted if `$var` is not numerical. Alternatively, a loop could be aborted (`break`) or the control-flow of a user-defined function ended (`return`).

```
1 if(!is_numeric($var))
2   die("not numeric.");
```

Listing 21: Path-sensitive program termination.

A static analysis tool should not only be aware of the fact that there is no jump from the `if`-block to the following code, but also that the conditional termination of the program prevents any non-numerical characters after the `if`-block. Considering more complex code and the *halting problem*, which proves the undecidability of all program halts with another program, it is evident that static code analysis cannot reason about all security mechanisms correctly.

2.5.3 Path-sensitive Validation

Another challenge for static analysis is path-sensitive usage of input validation. A typical example is given in Listing 22, where the variable `$error` is used to flag bad input.

```
1 if(!is_numeric($var)) {
2   $error = true;
3 }
4 if(!$error) { }
```

Listing 22: Path-sensitive validation.

The variable `$error` is independent from the variable `$var` that is analyzed for tainted input. Thus, its relevance for input validation is likely missed by path-insensitive static analysis. In contrast, analyzing all variables in all conditions of an execution path for input validation is very expensive for long paths and inter-procedural data flow.

3. STATIC ENUMERATION OF SECURITY MECHANISMS

Following the paradigm of *sources* and *sinks* (see Section 2), a web application can be audited for security vulnerabilities in an automated fashion with static code analysis (e.g., [1, 4, 12, 35]). Although dynamic code analysis provides a more precise data flow analysis for one execution path, static code analysis can (theoretically) achieve full code coverage and allows to enumerate all security mechanisms utilized in a given application. During static analysis, the application's source code is transformed into an abstract data model to analyze the data flow. A security vulnerability is reported if a source *flows* into a sensitive sink without any sanitization or validation in between. While sources and sinks can be easily configured, one of the main challenges for static analysis tools is to detect and evaluate these security mechanisms to avoid false vulnerability reports (*false positives*). In addition, we have shown that identifying the incorrect usage of security mechanisms is the key to detect vulnerabilities in modern applications [4]. Studying the occurrences of security mechanisms helps static analysis engineers to focus on the main root causes for false reports.

A erroneous approach to count security mechanisms in an application is to count the occurrences of security related operators and built-in functions in the code. On the one hand, this leads to an over-approximation when these features are used for other purposes than for data sanitization or validation, such as a type-cast of non-sensitive data. On the other hand, this leads to an under-approximation when these features are declared in reusable code once but called multiple times at runtime, such as a user-defined function.

A more precise enumeration of security mechanisms is achieved by leveraging static data flow and taint analysis. Here, the security relevance of data flow through such a mechanism can be evaluated by the taint status of the data. A mechanism should only be associated with security if it sanitizes or validates *tainted* data and this data reaches a sensitive sink. For this purpose, we modified the data flow and taint analysis of the existing prototype RIPS that is based on block and function summaries. It is able to simulate built-in security features precisely (for details, please refer to our paper [4]). In this section, we briefly recall the analysis steps of RIPS and introduce our modifications for enumeration, in order to answer our research questions **Q3**-**Q5**. Furthermore, we discuss limitations of our approach.

3.1 Data Flow Analysis

RIPS transforms every PHP file's code into an *abstract syntax tree* (AST). At jump nodes, the tree is split into *basic blocks*. These are connected by *block edges* that hold the branch's condition and build a *control flow graph* (CFG).

The data flow within each basic block is simulated by inferring memory locations and values from the AST to *data symbols* [4]. A data symbol is an abstract representation of data and used to store meta information, such as the data type, applied sanitization, encoding, or escaping. Examples for memory locations are variables, arrays, or object properties. Different types of memory locations are inferred to different types of symbols with additional meta information (see Section 3.2). Static strings are stored in a *value* symbol.

Data symbols assigned to a memory location are indexed in each basic block's *summary* by the location's name. The *block summary* stores the summarized data flow within one block. It is used to perform backwards-directed data flow analysis between multiple connected basic blocks throughout the CFG. By recursively looking up location names, data symbols can be resolved from previous basic blocks. Meta information, such as sanitization or encoding, is inherited from looked up symbols to resolved symbols.

Similarly, a *function summary* saves the data flow through a user-defined function and its side effects. The summary is reused for every function call context-sensitively, by adjusting call-site arguments to the summary.

3.2 Sanitization and Pitfall Tags

The symbols' meta information about sanitization is inferred from PHP operators and built-in functions within the AST. For this purpose, a whitelist of sanitizers is used and complex sanitizers are simulated carefully [4]. RIPS assigns different *sanitization tags* to the inferred data symbols that precisely point out the sanitized markup contexts.

For example, the built-in function `htmlentities()` sanitizes against *double quotes* and *lower-than* characters by default. Thus, the corresponding *sanitization tags* for double-quoted attributes or HTML elements are added to the argument's data symbol. If the mechanism sanitizes against all types of injection flaws, such as an integer typecast, a *universal* sanitization tag is added. In order to track multiple levels of escaping and encoding, each data symbol has an *escaping* and *encoding stack* that is modified by certain built-in functions, such as `addslashes()` and `urlencode()`. String and regular expression replacements are evaluated against a configured list of characters that are required for exploitation of specific markup contexts. If a character is replaced, the according sanitization tag is added to the data symbol.

For our study, we leverage these sanitization tags and extended our prototype by adding *pitfall tags*. These are inferred from the AST for delusive sanitization, such as the increment operator (see Section 2.1.2), or weak validation, such as fragmentary regular expressions (see Section 2.4.3).

3.3 Sanitization-centric Taint Analysis

During a basic block's simulation, taint analysis is invoked for sensitive arguments of sensitive sinks, such as the `echo` operator or `mysql_query()` built-in function. First, all strings are reconstructed that flow into the sink by using backwards-directed data flow analysis (see Section 3.1). Here, strings are fetched from *value* symbols and data symbols of sources are mapped to place holders. Unknown memory locations are resolved to empty strings. Then the place holders' positions within each string are analyzed to determine the markup context. If (according to the meta information) the data symbol of a place holder is not sanitized regarding the detected markup context, a vulnerability report is issued [4]. The report is *not* issued if the resolved data symbol possesses a matching sanitization tag.

For our study, we extended the taint analysis of our prototype to log such a successfully applied sanitization mechanism. For this purpose, we also added information about the used security mechanism when sanitization tags are applied to a data symbol by an operator or built-in function. Furthermore, we log a taken pitfall if a source's data symbol owns wrong sanitization tags, insecure encoding, or escaping regarding the detected markup context. A pitfall is logged if no correct sanitization tags are found but a *pitfall tag* was assigned to the data symbol.

3.4 Validation-centric Taint Analysis

In contrast to sanitization mechanisms, operators and built-in functions that perform a security related *validation* of their arguments are inferred from an AST to a *boolean* data symbol. It stores the information about the validated argument and the condition for successful validation (*true* or *false*). Additionally, the corresponding sanitization and pitfall tags are added for the type of validation. The *boolean* data symbol allows to track the flow of validated data in form of a *true* or *false* constant intra- and inter-procedurally.

Similarly to basic blocks, RIPS simulates block edges. If a *boolean* data symbol is inferred from the condition's AST, its sanitized data symbol is added to the block edge. During backwards-directed taint analysis of a data symbol, the sanitization and pitfall tags are copied from a block edge that validates the analyzed data symbol. At the end of the taint analysis, the collected sanitization and pitfall tags can be evaluated against the markup context. Due to our modification, successful validation leads to logging of the corresponding sanitization tags and erroneous validation issues a vulnerability report and logging of any pitfall tags.

3.5 Side Effects and Limitations

As a side effect, our modified prototype is significantly slower than the unmodified version. The unmodified version stops resolving data when it is validated by a block edge. Moreover, data symbols that were sanitized with a *universal* sanitization tag are not resolved further. Our modified version tries to fully resolve these symbols, however. Although exploitation is impossible, it tests if the symbol is resolved to a source in order to determine if the detected mechanism was applied for a security purpose.

Our prototype suffers from the limits of static code analysis. First of all, the detection of path-sensitive security mechanisms is unsound: although our prototype detects basic path-sensitive sanitization and termination, it fails to analyze complex forms of path-sensitive validation (see Section 2.5.3). Furthermore, our prototype is unable to handle custom template engines that combine data with file content. Lastly, complex string construction within loops, as it is sometimes used by SQL query builders, is handled imprecisely. These limitations can potentially lead to false negatives and false positives.

4. EMPIRICAL STUDY ON SECURITY MECHANISM USAGE

Based on our extended prototype, we empirically study the usage of different security mechanisms in combination with the markup context. This allows us to evaluate common and uncommon combinations, as well as, associated pitfalls. We first introduce our methodology of enumeration (see Section 4.1) and the software picked (see Section 4.2). Then we present our results in Section 4.3 that provides answers to **Q3-Q5** in detail and discuss our lessons learned in Section 4.4. Threats to validity are addressed in Section 4.5.

4.1 Methodology

We limit our study to XSS and SQL injection vulnerabilities, because these are the most prevalent injection flaws. Both types have a variety of contexts by using a separate markup language (namely HTML and SQL). A security mechanism is counted when a *unique* source flows into a *unique* markup context of a sink and was sanitized or validated correctly by a security mechanism. Incorrect sanitization or validation regarding the markup context is counted as pitfall, as well as any present pitfall tag. The backwards-directed taint analysis allows us to count only the nearest security mechanism before the sink. In the case of a path-sensitive sanitization or termination, the corresponding *validation* mechanism is counted (refer to Section 2.5). The detection of path-sensitive validation is disabled (reasons in Section 3.5). Furthermore, we excluded second-order tainting [5] and validation (see Section 2.3.7) because it depends on the analysis of *taintable* resources which is more error-prone than the detection of other security mechanisms.

Table 1: Overview of 25 selected applications with the amount of analyzed lines of code (LOC) and detected markup injections (HTML, SQL, or JavaScript).

Software	Version	LOC	HTML	SQL	JS
Beehive	1.4.3	105 325	2 976	402	0
CMSSimple	1.11.11	137 222	190	335	1
Concrete5	5.6.3.1	317 025	1 823	161	109
Couch CMS	1.4	37 073	25	29	16
e107	1.0.4	157 706	2 561	828	937
FluxBB	1.5.6	28 945	268	145	2
FreePBX	2.11.0.25	75 909	147	36	10
FUDForum	3.0.6-RC2	74 421	556	211	31
HotCRP	2.92	40 865	181	106	3
LiveZilla	5.2.0.1	43 593	40	181	0
Nucleus CMS	3.6.5	38 268	61	49	1
OpenConf	6.0	21 836	325	180	2
osCommerce	2.3.4	85 563	2 615	788	0
Phorum	5.2.19	73 841	304	699	2
PHP Fusion	7.02.07	54 584	805	563	40
PHP Nuke	8.3.2	200 767	261	291	0
phpList	3.0.6	103 647	670	169	15
Pligg CMS	2.0.1	62 588	24	258	0
PunBB	1.4.2	43 268	119	84	2
Roundcube	1.0.2	158 435	179	24	0
Serendipity	1.7.8	212 705	137	418	19
Squirrelmail	1.4.22	56 194	1 097	123	3
Wacko Wiki	5.4.0	103 217	100	132	0
Xoops	2.5.6	142 749	209	74	10
Zen Cart	1.5.1	131 458	1 029	1 810	5
Sum	n/a	2 507 204	16 702	8 096	1 208
Average	n/a	100 288	668	324	48

4.2 Corpus

In order to obtain the most precise results, we carefully chose the applications to analyze. First, we gathered a coarse list of PHP applications according to the following criteria:

- The application is open source, active, and popular according to W3Tech's usage statistic [29].
- The application has an size of at least 20 KLOC.
- The application works standalone and does not require additional code.

Then, we excluded applications from our list that make an extensive use of reflection or application frameworks. As mentioned in Section 3.5, static analysis of these components is limited and often requires manual configuration [26]. We discuss related threats to validity in Section 4.5. The list of our selected applications for evaluation is given in Table 1.

Table 2: Mechanisms safely applied to HTML contexts.

	DQ	Element	SQ	Comment	URL
Converting	3 367	839	18	8	10
Type Validation	1 816	1 494	15	27	0
Comparing	999	1 269	32	35	95
Explicit Typecast	439	1 075	34	57	20
Regex Validate	903	397	145	1	1
String Replace	27	513	517	3	1
Null Validation	167	205	9	15	0
Other	918	869	48	37	16
Sum	8 636	6 661	818	183	143

Table 3: Safe JS context.

	Script	Event
TypeVal	465	2
ExplCast	206	27
Compare	161	7
Replace	80	2
NullVal	31	4
RegexRepl	31	2
Encode	26	7
Other	75	17
Sum	1 075	68

Table 4: JS pitfalls.

	Script	Event
Converting	34	8
Replace	14	0
Compare	4	0
RegexCheck	3	0
RegexRepl	1	0
Truncate	1	0
Decode	0	0
Other	0	0
Sum	57	8

4.3 Results

In total, we analyzed 2.5 million lines of code and 26,006 unique data flows where a source *flows* sanitized (53%) or validated (47%) into a sensitive sink. Data flows without any applied security mechanism are excluded from our study.

As shown in Table 1, the most common markup context we found is HTML (64%), followed by SQL (31%) and JavaScript (5%). This is likely related to the fact that an average application prints more data to the response page than that it interacts with the database [9]. The `style` context (CSS) appeared rarely and was excluded from our study.

As a preliminary answer to research question **Q3**, we found that user input is primarily secured with a type validation (19%) or an explicit type cast (16%). The extensive use of type-related security mechanisms shows the additional work on the developer side that would otherwise be handled by default in a strongly typed language. Other security mechanisms are applied context-sensitively to the markup and are revealed throughout this section in order to answer research question **Q4**. Format validation or PHP's valuable filter functions are the least detected security mechanisms in our test set (<1%). Instead, string replacement and regular expressions are used, which are the security mechanisms with the highest pitfall density in our test corpus (**Q5**).

4.3.1 HTML Markup Security

Table 2 shows the seven most frequently used security mechanisms to secure the five most common HTML markup contexts, according to our study. For example, 3,367 occasions were detected where user input within a double-quoted (*DQ*) HTML attribute was correctly sanitized by converting characters. Table 5 shows frequent markup contexts where this mechanism failed, for example for 3 URL attributes.

In summary, the HTML landscape is dominated by double-quoted HTML attributes (52%) and the context between two HTML elements (41%), followed by single-quoted (*SQ*) attributes context (5%). HTML comments, fully controlled URL attributes, and attribute names are rare contexts (<1%).

Table 5: Pitfalls triggered in HTML contexts.

	Element	DQ	SQ	URL	Comment
String Replace	57	12	3	2	1
Regex Validation	33	10	6	0	0
Escaping	40	0	0	0	0
Regex Replace	20	11	4	0	2
Comparing	7	18	8	1	0
Converting	0	0	6	3	0
String Search	4	2	0	0	0
Other	2	9	0	0	0
Sum	163	62	27	6	3

The most frequently used security mechanism is an HTML character conversion by PHP built-in functions, such as `htmlentities()` (see Table 2 and Table 5, *cursive*). This data sanitization is applied to 25% of all 16,702 detected HTML markup contexts, primarily to the two markup contexts it is designed for: double-quoted attributes and between HTML elements. However, we also detected 24 cases where it is applied to a single-quoted attribute context that requires an additional parameter to the `htmlentities()` function (see Section 2.2.1). In 6 cases (e. g., in FreePBX), this pitfall was triggered which leads to the suggestion to always use double-quoted HTML attributes. Furthermore, type validation (20%) and explicit type casting (10%) is regularly applied, followed by data validation with regular expressions (9%) and data sanitization with string replacements (7%).

Especially the two latter are prone to pitfalls, as shown **highlighted** in Table 5. We found that 7% of all applied string replacements and 3% of all applied regular expressions to tainted data are insufficiently sanitizing or validating the HTML context. A single forgotten character that slips through the string replacement or regular expression filter leads to an exploitable vulnerability. Moreover, cross-site scripting occurs when data is sanitized for a *different* vulnerability type. For example, in 40 occasions data is correctly escaped for the use in a SQL query, but later printed insecurely to the HTML markup between two elements.

4.3.2 JavaScript Markup Security

As shown in Table 3, the prevalent JavaScript context in our selected applications is the HTML `script` tag that clearly dominates over eventhandler attributes (*Event*). The table lists the top seven security mechanisms applied to both contexts, as detected by our prototype in the corpus. Next to type-related security mechanisms, such as type validation (39%) and explicit type casting (19%), many custom sanitization approaches are found that base on string comparison (14%), regular expressions (3%), or string replacement (8%).

Similar to the HTML markup, the latter is the root cause for 14 pitfalls (see Table 4, **highlighted**). For example, in Couch CMS the backslash and double-quote character is replaced to prevent an outbreak of double-quotes. However, an attacker can terminate the current `script` tag and start a new JavaScript context that requires no quotes by injecting `</script><script>`. According to our study, the most commonly taken pitfall for a JavaScript context is based on character conversion. In fact, we found more vulnerable applications of character conversion to a JavaScript context than secure ones. This is related to the fact that most JavaScript contexts use no quotes or single-quotes, which are not converted by the built-in functions by default.

Table 6: Safe SQL contexts.

	SQ	NQ	DQ
ExplCast	1 140	1 028	1
Escape	1 519	0	38
TypeVal	826	287	6
Compare	537	248	19
RegexVal	497	77	4
Replace	382	39	4
Prepare	0	352	0
Other	489	316	49
Sum	5 390	2 347	121

Table 7: SQL pitfalls.

	NQ	SQ	DQ
Escape	72	0	0
RegexVal	26	11	0
Replace	18	15	2
Compare	15	12	0
Truncate	7	20	0
Prepare	21	2	0
RegexRepl	9	1	0
Other	2	4	0
Sum	170	65	2

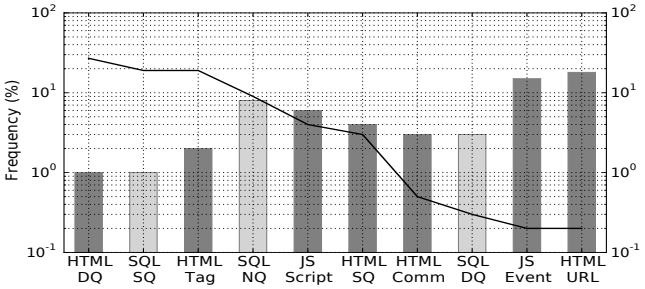

Figure 1: Pitfall density (bars) versus frequency (line) of markup contexts prone to XSS (dark) and SQLi (light).

4.3.3 SQL Markup Security

Table 6 and Table 7 compare the distribution of correctly and wrongly applied security mechanisms to 8,096 detected SQL contexts. In our evaluation, 67% of the sources are embedded into single-quotes (SQ), 31% without quotes (NQ), and 2% into double-quotes (DQ). The values are usually sanitized (27%) or validated (14%) by data type, or sanitized by escaping (19%). However, for 72 escapes (4%), the surrounding quotes are forgotten which leads to SQL injection. This is the most frequent pitfall encountered for SQL markup. Contrarily, we found that only 5% of all SQL queries in our selected applications use prepared statements which would prevent these obstacles. Moreover, 7% of all these prepared statements are handled unsafe (e. g. in PunBB), indicating that the concept is not thoroughly understood. As for the HTML and JavaScript markup, many vulnerabilities stem from insufficient string replacement or regular expression validation (see Table 7, **highlighted**). While these mostly recognize quotes within the input, the SQL markup misses quotes or the backslash character is forgotten.

4.4 Lessons Learned

Based on our results, we were able to suggest answers to our research questions regarding the diversity of security mechanisms and pitfalls (**Q3-Q5**). This provides valuable insight for the practice and teaches the following lessons.

First of all, we learned about pitfall-prone markup context. In order to find vulnerable code as a developer, code auditor, or static analysis engineer, an increased focus can be applied to markup contexts with a high *pitfall density* (detected pitfalls per detected markup contexts) and high frequency (see Figure 1). According to our evaluation, fully controlled URL attributes (18%) and eventhandlers (15%) are highly prone to pitfalls. However, these appear seldom in code (0.2% of all detected markup contexts). More commonly in practice are SQL values with no quotes (9%), `script` tags (4%), and single-quoted HTML attributes (3%) that have a pitfall density of 8%, 6%, and 4%, respectively (according to our analysis results). These are the contexts to keep an eye on. Least likely affected by pitfalls are double-quoted HTML attributes and single-quoted SQL values (1%).

Second, we learned about pitfall-prone security mechanisms. As **highlighted** throughout Table 2-7, custom security mechanisms based on regular expressions and string replacement are more prone to pitfalls than other security mechanisms, but appear rather frequently. These should be carefully inspected, and often can be replaced with a less error-prone security mechanism. We also believe that a high pitfall density for JavaScript markup is related to the fact that there is no designated built-in function in PHP.

Last but not least, we use the lessons learned as a new metric for our tool to improve the severity ranking of vulnerability reports. Except for the *universal* sanitization tag that introduces performance issues (see Section 3.5), we kept the logging of applied security mechanisms in our prototype. This allows us to rank vulnerabilities detected in error-prone markup contexts, such as an XSS within an URL attribute, higher than reports in other markups that are statistically less vulnerable. Similarly, reported vulnerabilities with a pitfall in string replacement or a regular expression are ranked higher. The logging also helps us to validate our true negative rate and to detect a low code coverage. We assume that a large application will result in a certain amount of logged security mechanisms and markup contexts. Otherwise, this indicates a low code coverage that could possibly stem from a template engine or a SQL builder.

4.5 Threats to Validity

There are certain threats to validity of our results that caution to draw strong conclusions and to generalize. Our corpus consists of only 25 popular applications. We excluded some popular applications, such as Wordpress, Joomla, and Drupal, because for these highly dynamic applications, a low code coverage of our static analysis tool is expected, which could tamper the results. We refrained from adding more unpopular applications because we believe that they are more vulnerable. Although more studies need to be conducted to verify our results for more applications, we believe that our corpus represents a comprehensive set of modern and popular applications in order to provide reasonable indicators. However, a different corpus may introduce a different amount of security mechanisms and pitfalls and the detection rate of our tool may be different for each application.

Moreover, as discussed in Section 3.5, static code analysis is limited. Although we would like to find all vulnerabilities present in an application, static analysis allows only to detect a fraction. While we experienced good results with our prototype in the past and verified random samples, we cannot guarantee the absence of false positives, false negatives, or mistakes. We tried to mitigate this threat by excluding applications not suited for our prototype and the results appear to be reasonable according to our experience with security vulnerabilities in PHP applications [4–6].

Finally, our analysis cannot reason about the intention of the developer. This poses a threat to the evaluation of security mechanisms and pitfalls because, for example in case of a string comparison, data can be accidentally validated or a pitfall might be triggered although no validation was intended in first place.

5. RELATED WORK

Input sanitization and validation mechanisms became very important for software developers with the raise of string-related vulnerabilities, especially in the context of web applications and vulnerabilities such as XSS and SQL injection. In this section, we review related empirical studies and automated analysis of security mechanisms in (PHP-based web) applications.

5.1 Empirical Studies

Next to security vulnerability trends [3, 7, 28, 31], empirical studies regarding sanitization and validation approaches in PHP applications were conducted. However, the covered mechanisms are either incomplete or studied in a different context. This leads to different results because, for example, not every string comparison is a security mechanism.

Hills et al. conducted an interesting study of the usage of PHP features in 19 applications [9]. Among different features, the occurrences of type casts and binary operations were studied. However, these features were not interpreted regarding security and no other security mechanisms were covered. Scholte et al. empirically analyzed the data type of the parameters that are affected by XSS and SQL injection for over 7,000 vulnerabilities [23]. Next to native data types, such as string, integer, and boolean, they also considered custom data types, such as *email*, *url*, or *username*. As a result, the most commonly affected data types were identified, as well as the lack of built-in sanitization mechanisms for these types in common web frameworks.

Weinberger et al. empirically studied present sanitization approaches against XSS in web application frameworks [34]. They analyzed the availability of sanitization approaches for different HTML markup contexts for five PHP frameworks. Furthermore, eight PHP applications were studied for the usage of various markup contexts. A templating framework was proposed by Samuel et al. that uses type qualifiers to automate context-sensitive XSS sanitization [19]. Our analysis is more comprehensive and extends these studies.

In his dissertation about decision procedures for string constraints, Hooimeijer studied the occurrence of 113 PHP built-in string functions in 88 applications [10]. Among these functions are length limiting, regular expression, and formatting functions, but these are not interpreted regarding sanitization or validation. Saxena et al. developed *Script-Gard* to detect and correct the misplacement of sanitizers in ASP.NET applications with dynamic analysis [22]. They studied one application with 400 KLOC for context-mismatched sanitization or sanitizer sequences. In comparison, we studied 25 applications and 2.5 million LOC.

5.2 Static Security Mechanism Analysis

A variety of static analysis approaches have been proposed to automatically identify security vulnerabilities in PHP applications based on insufficient sanitization and validation.

Zheng and Zhang introduced path-sensitive static analysis for PHP applications with Z3-str [38]. They leverage a modified version of the Z3 SMT solver that is also capable of analyzing strings. Shar and Tan proposed static code attributes for predicting SQLi and XSS vulnerabilities [24, 25]. Among their attribute vectors are six validation and six sanitization mechanisms. Other security mechanisms introduced by our work are missed and likely lead to false positives.

Yu et al. built an automata-based string analysis tool called *STRANGER* [36] based on the static code analysis tool Pixy [12]. STRANGER detects security vulnerabilities in PHP applications by computing possible string values using a symbolic automata representation of common string functions, including escaping and replacement functions. Later, they automatically generated sanitization statements for detected vulnerabilities by using regular expression replacements [37]. Balzarotti et al. combine static and dynamic analysis techniques to identify faulty custom sanitization routines [1]. The static analysis component of their tool called *Saner* extends Pixy and analyzes string modification with automata, while the dynamic component verifies analysis results to reduce false positives. None of these tools detects input validation.

Kneuss et al. developed a static analyzer called *PHANTM* to detect type errors in PHP applications [14]. Although not primarily in focus, they detected security vulnerabilities during their evaluation. Wasserman and Su leverage string analysis with context free grammars to detect XSS [33] and SQLi [32] vulnerabilities based on insufficiently-checked untrusted data. They cover string replacement and escaping, while path-sensitive input validation leads to false positives. Minamide developed a string analyzer to approximate the output of PHP applications using a context-free grammar [16]. It models a variety of sanitization functions but can only prove the absence of predefined attack vectors.

Sanitization analysis was also applied to other programming languages, such as JavaScript [20, 21], Java [27], and ASP.NET [11, 15, 22]. Our work complements such studies and provides a comprehensive overview of sanitization approaches used in modern web applications.

6. CONCLUSION

In PHP, a variety of security mechanisms exist that can be applied by developers in order to defuse user input for sensitive operations. However, different markup contexts of different operations require different security mechanisms.

In this paper, we empirically analyzed how developers utilize data sanitization and validation mechanisms in practice to prohibit malicious input. We discussed typical programming patterns we experienced during analysis of a variety of PHP-based web applications. Furthermore, we presented a static code analysis approach to detect and study these security mechanisms and extended our prototype. By analyzing 25 popular PHP applications, we obtained valuable insights into common pitfalls and security vulnerabilities. For example, we found evidence that single-quoted HTML attributes are more likely causing a vulnerability than double-quoted.

Our results help us (and other static analysis engineers) to focus on the detection and precise simulation of the most common security mechanisms. We expect that (web) applications implemented in other languages contain similar programming patterns that developers, code auditors, and static analysis engineers need to be aware of. Furthermore, our results serve as a metric to rank vulnerability reports and to verify the code coverage in our tool.

Threats to validity of our results exist, that in the worst case allow us to only generate hypotheses. In the future, we plan to work on the support for frameworks in order to evaluate even more applications. More specifically, template engines and SQL builders can lead to imprecision in our static analysis that we want to eliminate.

7. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[2] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-side XSS Filters. In *Conference on the World Wide Web (WWW)*, 2010.

[3] S. Christey and R. A. Martin. Vulnerability Type Distributions in CVE, May 2007.

[4] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[5] J. Dahse and T. Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2014.

[6] J. Dahse, N. Krein, and T. Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[7] M. Doyle and J. Walden. An Empirical Study of the Evolution of PHP Web Application Security. In *Security Measurements and Metrics (Metrisec)*, 2011.

[8] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, 2006.

[9] M. Hills, P. Klint, and J. Vinju. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

[10] P. Hooimeijer. Decision Procedures for String Constraints. *Ph.D. Dissertation, University of Virginia*, 2010.

[11] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*, 2011.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy (S&P)*, 2006.

[13] A. Klein. Cross-Site Scripting Explained. *Sanctum White Paper*, 2002.

[14] E. Kneuss, P. Suter, and V. Kuncak. Phantm: PHP Analyzer for Type Mismatch. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2010.

[15] B. Livshits and S. Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013.

[16] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Conference on the World Wide Web (WWW)*, 2005.

[17] OWASP. OWASP Secure Coding Practices. `https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide`, as of January 2015.

[18] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A Large Scale Study of Programming Languages and Code Quality in Github. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[19] M. Samuel, P. Saxena, and D. Song. Context-sensitive Auto-sanitization in Web Templating Languages using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 587–600, 2011.

[20] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for Javascript. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[21] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Symposium on Network and Distributed System Security (NDSS)*, 2010.

[22] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[23] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *ACM Symposium On Applied Computing (SAC)*, 2012.

[24] L. K. Shar and H. B. K. Tan. Predicting Common Web Application Vulnerabilities from Input Validation and Sanitization Code Patterns. In *Automated Software Engineering (ASE)*, 2012.

[25] L. K. Shar, H. B. K. Tan, and L. C. Briand. Mining SQL Injection and Cross Site Scripting Vulnerabilities using Hybrid Program Analysis. In *International Conference on Software Engineering (ICSE)*, 2013.

[26] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint Analysis of Framework-based Web Applications. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[28] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[29] W3Techs. Usage of Content Management Systems for Websites. `http://w3techs.com/technologies/overview/content_management/all`, as of January 2015.

[30] W3Techs. Usage of Server-side Programming Languages for Websites. `http://w3techs.com/technologies/overview/programming_language/all`, as of January 2015.

[31] J. Walden, M. Doyle, G. A. Welch, and M. Whelan. Security of Open Source Web Applications. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2009.

[32] G. Wasserman and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[33] G. Wasserman and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)*, 2008.

[34] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[35] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, 2006.

[36] F. Yu, M. Alkhalaf, and T. Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[37] F. Yu, M. Alkhalaf, and T. Bultan. Patching Vulnerabilities with Sanitization Synthesis. In *International Conference on Software Engineering (ICSE)*, 2011.

[38] Y. Zheng and X. Zhang. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In *International Conference on Software Engineering (ICSE)*, 2013.