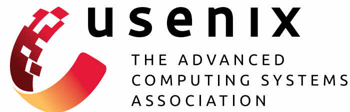

Static Detection of Second-Order Vulnerabilities in Web Applications

Johannes Dahse and Thorsten Holz, *Ruhr-University Bochum*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dahse

This paper is included in the Proceedings of the
23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of
the 23rd USENIX Security Symposium
is sponsored by USENIX

Static Detection of Second-Order Vulnerabilities in Web Applications

Johannes Dahse
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
johannes.dahse@rub.de

Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
thorsten.holz@rub.de

Abstract

Web applications evolved in the last decades from simple scripts to multi-functional applications. Such complex web applications are prone to different types of security vulnerabilities that lead to data leakage or a compromise of the underlying web server. So called *second-order vulnerabilities* occur when an attack payload is first stored by the application on the web server and then later on used in a security-critical operation.

In this paper, we introduce the first automated static code analysis approach to detect second-order vulnerabilities and related multi-step exploits in web applications. By analyzing reads and writes to memory locations of the web server, we are able to identify unsanitized data flows by connecting input and output points of data in *persistent data stores* such as databases or session data. As a result, we identified 159 second-order vulnerabilities in six popular web applications such as the conference management systems *HotCRP* and *OpenConf*. Moreover, the analysis of web applications evaluated in related work revealed that we are able to detect several critical vulnerabilities previously missed.

1 Introduction

Web applications are the driving force behind the modern Web since they enable all the services with which users interact. Often, such applications handle large amounts of (potentially sensitive) data such as text messages, information about users, or login credentials that need to be stored persistently on the underlying web server. Further, *sessions* are used to temporarily store data about a user interacting with the web application during multi-step processes. All of this data can potentially be abused by an attacker to cause harm. Many different kinds of attacks against web applications such as *Cross-Site Scripting* (XSS) or *SQL injection* (SQLi) attacks are known and common injection flaws are well understood. Such attacks can be prevented by sanitizing user input and many approaches to address this problem were presented in the last few years (e.g., [2, 8, 15, 21, 22, 24, 27, 29]).

One common assumption underlying many detection and prevention approaches is that data that is already stored on the server is safe. However, an adversary might be able to bypass the defenses via so called *second-order vulnerabilities* if she manages to first abuse the web application to store the attack payload on the web server, and then later on use this payload in a security-critical operation. Such vulnerabilities are often overlooked, but they can have a severe impact in practice. For example, XSS attacks that target the application's users are worse if the payload is stored in a shared resource and distributed to all users. Furthermore, within *multi-step exploits* a vulnerability can be escalated to a more severe vulnerability. Thus, detecting second-order vulnerabilities is crucial to improve the security of web applications.

Detecting Second-Order Vulnerabilities To prevent such attacks, the source code of a given web application is assessed before it is deployed on a web server. This can be done either via dynamic or static analysis. There are several dynamic approaches to detect second-order XSS attacks via fuzzing [14, 19]. Generally speaking, such approaches try to inject random strings to all possible POST request parameters in a black-box approach. In a second step, the analysis tools determine if the random string is printed by the application again without another submission, indicating that it was stored on the web server. However, the detection accuracy for second-order vulnerabilities is either unsatisfying or such vulnerabilities are missed completely [4, 7, 13, 23]. Artzi et. al. [1] presented a dynamic code analysis tool that considers persistent data in sessions, but their approach misses other frequently used data stores such as databases or files. Furthermore, one general drawback of dynamic approaches is the typically low code coverage.

Static code analysis is a commonly used technique to find security weaknesses in source code. Taint analysis and similar code analysis techniques are used to study the data flow of untrusted (also called *tainted*) data into critical operations of the application. However, web applications can also store untrusted data to external resources and later on access and reuse it, a problem that is over-

looked in existing approaches. Since the data flow is deferred and can be split among different files and functions of the application, second-order vulnerabilities are difficult to detect when analyzing the source code statically. Furthermore, static code analysis has no access to the external resources used by the application and does not know the data that is stored in these.

We are not aware of any plain *static* code analysis implementation handling second-order vulnerabilities. The main problem is to decide whether data fetched from persistent stores is tainted or not. Assuming all data to be tainted would lead to a high number of false positives, while a conservative analysis might miss vulnerabilities.

Our Approach In this paper, we introduce a refined type of taint analysis. During our data flow analysis, we collect all locations in persistent stores that are written to and can be controlled (tainted) by an adversary. If data is read from a persistent data store, the decision if the data is tainted or not is delayed to the end of the analysis. Eventually, when all taintable writings to persistent stores are known, the delayed decisions are made to detect second-order vulnerabilities. The intricacies of identifying the exact location within the persistent store the data is written to is approached with string analysis. Furthermore, sanitization through database lookups or checks for existing file names are recognized.

We implemented our approach in a prototype for static PHP code analysis since PHP is the most popular server-side scripting language on the Web with an increasing market share of 81.8% [28]. Note that our approach can be generalized to static code analysis of other languages by applying the techniques introduced in this paper to the data flow analysis of another language. We evaluated our approach by analyzing six popular real-world applications, including *OpenConf*, *HotCRP*, and *osCommerce*. Overall, we detected and reported 159 previously unknown second-order vulnerabilities such as *remote command execution* vulnerabilities in *osCommerce* and *OpenConf*. We also analyzed three web applications that were used during the evaluation of prior work in this area and found that previous work missed several second-order vulnerabilities, indicating that existing approaches do not handle such vulnerabilities correctly.

In summary, we make the following three contributions:

- We are the first to propose an automated approach to statically analyze second-order data flows through databases, file names, and session variables using string analysis. This enables us to detect second-order and multi-step exploitation vulnerabilities in web applications.
- We study the problem of second-order sanitization, a crucial step to lower the number of potential false positives and negatives.
- We built a prototype of the proposed approach and evaluate second-order data flows of six real-world web applications. As a result, we detect 159 previously unknown vulnerabilities ranging from XSS to *remote code execution* attacks.

2 Technical Background

In this section, we introduce the nature of second-order vulnerabilities and multi-step exploits. First, we examine data flow through persistent data stores and the difficulties of analyzing such flows statically. We then present two second-order vulnerabilities as motivating examples.

2.1 Persistent Data Stores

We define *persistent data stores* (PDS) as memory locations that are used by an application to store data. This data is available after the incoming request was parsed and can be accessed later on by the same application to reuse the data. The term *persistent* refers to the fact that data is stored on the web server's hard drive, although it can be frequently deleted or updated. Note that this definition also includes session data since information about a user's session is stored on the server and can be reused by an adversary. We now introduce three commonly used PDS by web applications.

2.1.1 Databases

Databases are the most popular form of PDS found in today's web applications. A database server typically maintains several databases that consist of multiple tables. A table is structured in columns that have a specific data type and length associated with them. Stored data is accessed via SQL queries that allow to filter, sort, or intersect data on retrieval. In PHP, an API for database interaction is bundled as a PHP extension that provides several built-in functions for the database connection, and the query and access of data.

In contrast to other PDS, *writing* and *reading* to a memory location is performed via the same built-in query function. SQL has different syntactical forms of writing data to a table. Listing 1 shows three different ways to perform the same query.

```
1 // specified write
2 INSERT INTO users (id,name,pass) VALUES (1,'admin','foo')
3 INSERT INTO users SET id = 1, name = 'admin', pass = 'foo'
4 // unspecified write
5 INSERT INTO users VALUES (1,'admin','foo')
```

Listing 1: Writing to the database table *users* in SQL.

While the first two queries explicitly define the column names, the third query does not. We refer to the first type as *specified write* and to the second type as *unspecified write*. Both types convey a difficulty for static

analysis of the query: a *specified write* reveals the column names where data is written to, but does not reveal if there are any other columns in the table that are filled with default values. This hinders the reconstruction of table structures when analyzing SQL queries of an application statically. An *unspecified write* tells us exactly how many columns exist, but does not reveal its names. When the columns are accessed later on by name, it is unclear which column was filled with which value. The same applies for read operations. A *specified read* reveals the accessed column names in a field list, whereas an *unspecified read*, indicated by an *asterisk* character, selects all available columns without naming them.

In PHP, the queried data is stored as a result resource. There are different ways to fetch the data from the result resource with built-in functions, as shown in Listing 2.

```
1 // numeric fetch
2 $row = mysql_fetch_row($res);      echo $row[1];
3 // associative fetch
4 $row = mysql_fetch_assoc($res);    echo $row["name"];
5 $row = mysql_fetch_object($res);   echo $row->name;
```

Listing 2: Fetching data from a database result resource.

Basically, *numeric* and *associative fetch* operations exist. The first method stores the data in a numerically indexed array where the index refers to the order of the selected columns. The *associative fetch* stores the data in an array indexed by column name. It is also possible to store the data in an object where the property names equals the column names. The key difference is that the *associative fetch* reveals the accessed column names while the *numeric fetch* does not.

All different combinations of writing, reading, and accessing data can occur within a web application. In certain combinations, it is not clear which columns are accessed without knowledge about the database schema. For example, when data is written *unspecified* and fetched associatively. In practice, however, we are often able to reconstruct the database schema from the source code (see Section 3.4.1 for details).

2.1.2 Session Data

A common way of dealing with the state-less HTTP protocol are *sessions*. In PHP, the `$_SESSION` array provides an abstract way of handling session data that is stored within files (default) or databases. A session value is associated with an alphanumerical key that represents the memory location. Note that the `$_SESSION` array needs to be treated like any other superglobal array in PHP and it can be accessed in any context of the application. As any other array, it can be accessed and modified dynamically, inter-procedurally, and it can have multiple key names. Besides the `$_SESSION` array and the deprecated `$HTTP_SESSION_VARS` array, the built-in functions `session_register()` and `session_decode()` can be used to set session data.

2.1.3 File Names

A common source for vulnerabilities is an unsanitized file name. Developers often overlook that the file name of an uploaded file can contain malicious characters and thus can be used as a PDS for an attack payload. For example, Unix file systems allow any special characters in file names, except for the slash and the null byte [12]. NTFS allows characters such as the single quote that can be used for exploitation [20]. For detecting second-order vulnerabilities, we need to determine paths where files with arbitrary names are located. The analysis of a file upload reveals to which path a file is written to and if the file is named as specified by the user. In PHP, a file that is submitted via a multi-part POST request is stored in a temporary directory with a temporary file name. The temporary and original file name is accessible in the superglobal `$_FILES` array. Furthermore, built-in functions such as `rename()` and `copy()` can be used by an application to rename a file on the server. Note that also directory names can be used as PDS, for example when created with the built-in function `mkdir()`.

2.1.4 Excluded PDS

There are less popular PDS that we do not include in our analysis. For example, data can be retrieved from a CGI environment variable, a configuration file, or from an external resource such as an FTP or SMTP server [5]. However, these PDS are used rarely in practice and decisions can only be made with preconfigured whitelists. We only consider PDS that are tainted by the application itself and not through a different channel. Analyzing the data flow through file content will be an interesting addition in the future. Here, the challenge is to determine to what part of a given file data is written to and from what part of the file data is read from because the structure of the data within the file is unknown.

Note that data stored via PHP's built-in functions `ini_set()` or `putenv()` only exists for the duration of the current request. At the end of the request, the environment is restored to its original state. Thus, they do not hold to our definition of a PDS.

2.2 Second-Order Vulnerabilities

A taint-style vulnerability occurs if data controlled by an attacker is used in a security-critical operation. In the data flow model, this corresponds to tainted data literally flowing into a sensitive sink within one possible data flow of the application. We classify a second-order vulnerability as a taint-style vulnerability where the data flows through one or more PDS. Here, the attack payload is first stored in a PDS and later retrieved and used in a sensitive sink. Thus, *two* distinct data flows require analysis: (i) source to PDS and (ii) PDS to sink.

In the following, we introduce two motivating examples with a payload stored in a PDS. In general, every combination of a source, sensitive sink, and a PDS is possible. Depending on the application's design, the flow of malicious data occurs within a single or multiple attack requests (e.g., when different requests for writing and reading are necessary). Finally, we introduce multi-step exploits as a subclass of second-order vulnerabilities.

2.2.1 Persistent Cross-Site Scripting

Cross-Site Scripting (XSS) [16] is the most common security vulnerability in web applications [22]. It occurs when user input is reflected to the HTML result of the application in an unsanitized way. It is then possible to inject arbitrary HTML markup into the response page that is rendered by the client's browser. An attacker can abuse this behavior by embedding malicious code into the response that for example locally defaces the web site or steals cookie information.

We speak of *Persistent Cross-Site Scripting* if the attacker's payload is stored in a PDS first, read by the application again, and printed to the response page. In contrast to *non-persistent* (*reflected*) XSS, the attacker does not have to craft a suspicious link and send it to a victim. Instead, all users of the application that visit the affected page are attacked automatically, making the vulnerability more severe. Furthermore, a *persistent XSS* vulnerability can be abused to spread an XSS worm [18, 26].

Listing 3 depicts an example of a *persistent XSS* vulnerability. The simplified code allows to submit a new comment which is stored in the table `comments` together with the name of the author. If no new comment is submitted, it lists all previously submitted comments that are fetched from the database. While the comment itself is sanitized in line 7 by the built-in function `htmlentities()` that encodes HTML control characters, the author's name is not sanitized in line 6 and thus affected by XSS. Note that if the source code is analyzed top-down, it is unknown at the point of the `SELECT` query if malicious data can be inserted into the table `comments` by an adversary.

```
1  if(empty($_POST['submit'])) {
2     // list comments
3     $res = mysql_query("SELECT author,text FROM comments");
4     foreach(mysql_fetch_row($res) as $row) {
5        $comment = mysql_fetch_array($row);
6        echo $comment['author'] . ': ' .
7        htmlentities($comment['text']) . "<br />";
8     }
9  }
10 else {
11    // add comment
12    $author = addslashes($_POST['name']);
13    $text = addslashes($_POST['comment']);
14    mysql_query("INSERT INTO comments (author, text)
15                 VALUES ('$author', '$text')");
16 }
```

Listing 3: Example for *second-order XSS* vulnerability.

2.2.2 Second-Order SQL Injection

A *SQL injection* (SQLi) [9] vulnerability occurs when a web application dynamically generates a SQL query with unsanitized user input. Here, an attacker can potentially inject her own SQL syntax to arbitrarily modify the query. Depending on the environment, the attacker can potentially extract sensitive data from the database, modify data, or compromise the web server.

In Listing 4, user supplied credentials are checked in line 6. If the credentials are valid, the session key `loggedin` is set to *true* and the user-supplied user name is saved into the session key `user`. In case the user-supplied data is invalid, the failed login attempt is logged to the database with the help of the user-defined `log()` function. Here, a *second-order SQLi* occurs: if an attacker registers with a malicious user name, this name is written to the session key `user` and on a second failed login attempt used in the logging SQL query.

```
1  function log($error) {
2     $user = $_SESSION['user'];
3     mysql_query("INSERT INTO logs (error, user)
4                  VALUES ('$error', '$user')");
5  }
6  if(validAuth($_POST['user'], $_POST['pass']) {
7     $_SESSION['loggedIn'] = true;
8     $_SESSION['user']     = $_POST['user'];
9  }
10 else {
11    log('Failed login attempt');
12 }
```

Listing 4: Example for *second-order SQLi* vulnerability.

2.2.3 Multi-Step Exploitation

Within a second-order vulnerability, the first order (e. g., safe writing of user input into the database or a file path) is not a vulnerability by itself. However, *unsafe* writing can lead to other vulnerabilities. We define a multi-step exploit as the exploitation of a vulnerability in the second order that requires the exploitation of an unsafe writing in the first order. Thus, a multi-step exploit is a subclass of a second-order vulnerability and it can drastically raise the severity of the first vulnerability.

Since we only consider databases, sessions, and file names as PDS in our analysis, the following vulnerabilities are relevant:

- *SQLi*: A SQLi in an `INSERT` or `UPDATE` statement leads to a full compromise of all columns in the specified table. Furthermore, a SQLi in a `SELECT` query allows arbitrary data to be returned.
- *Path traversal*: A *path traversal* vulnerability allows to change the current directory of a file operation to another location. Arbitrary file names can be created in arbitrary locations if a *path traversal* vulnerability affects the renaming or creation of files.
- *Arbitrary file write*: An *arbitrary file write* vulnerability can modify or create a new session file, leading to the compromise of all session values.

3 Detecting Second-Order Vulnerabilities

In the following, we describe our approach to automatically detect second-order vulnerabilities via static code analysis. For this purpose, we extended our prototype *RIPS* [6] that uses *block summaries* [30]. In this section, we first briefly review the used data flow and taint analysis approach of RIPS (Sections 3.1 and 3.2). Afterwards, we explain our novel additions for detecting second-order vulnerabilities and multi-step exploits (Sections 3.3–3.5).

3.1 Data Flow Analysis

RIPS leverages a context-sensitive, intra- and inter-procedural data flow analysis. We use basic block, function, and file *summaries* [30] for efficient, backwards-directed data flow analysis [6]. First, for each PHP file's code, a *control flow graph* (CFG) consisting of connected basic blocks is generated. Definitions of functions, classes, and methods within the code are extracted. Then, every CFG is analyzed top-down by simulating the connected basic blocks one by one. A block edge that links two connected basic blocks is simulated as well to identify data sanitization.

During the simulation of one basic block, all assigned data is transformed into *data symbols* that we will introduce later. The flow of the data is inferred from these symbols and summarized in a *block summary* [30] that maps data locations to assigned data. The return results and side-effects (e.g., data assignment or sanitization) of called built-in functions are determined by a precise simulation of over 900 unique functions.

If a user-defined function is called within a basic block, its CFG is generated and all basic blocks are simulated. Based on these block's summaries, the data flow within the function is determined by analyzing `return` statements in a similar way to taint analysis (see Section 3.2). The results are stored in a *function summary*. This summary is used for each call of the user-defined function, while return values, global variables, and parameters are adjusted to the callee's arguments and environment context-sensitively. When all basic blocks of a file's CFG are simulated, a *file summary* is generated in a similar way to functions that is used during file inclusion.

Data and its access within the application's code is modeled by so called *data symbols* [6]:

- `Value` represents a static `"string"`, integer, float, or a resolved `CONSTANT`'s value. Defined constant values are stored in the environment.
- `Variable` represents a `$variable` by its name.
- `ArrayDimFetch` represents the access of an array (`$array[k]`) and extends the `Variable` symbol with a *dimension* (`k`). The dimension lists the fetched array keys in form of data symbols.
- `ArrayDimTree` represents a newly declared array or the assignment of data to one array key (`$array[k] = $data`). It is organized in a tree structure. The array keys are represented by array edges that point to the assigned data symbol. The `ArrayDimTree` symbol provides methods to add or fetch symbols by a *dimension* that is compared to the tree's edges.
- `ValueConcat` represents the concatenation of two or more data symbols (`$a.$b`). Two consecutive `Value` symbols are merged to one `Value` symbol.
- `Multiple` is a container for several data symbols. It is used, for example, when a function returns different values depending on the control flow or PHP's *ternary* operator is used (`$c ? $a : $b`).

During data flow analysis, one or more *sanitization tags* can be added to a data symbol, for example if sanitization is applied by built-in functions such as `addslashes()` or `htmlentities()`. Each sanitization tag represents one context, for example, a *single-quoted SQL value* or a *double-quoted HTML attribute*. A symbol can be sanitized against one context, but be vulnerable to another. The tags are removed again when built-in functions such as `stripslashes()` or `html_entity_decode()` are called. Furthermore, information about encoding is added to every data symbol.

3.2 Context-Sensitive Taint Analysis

The goal is to create a vulnerability report, whenever a tainted data symbol δ flows into a sensitive sink. Our implementation is performed with 355 sensitive built-in functions of PHP. If a call to a sink is encountered during block simulation, its relevant arguments are analyzed. First, the argument is transformed into a data symbol. If the symbol was defined within the same basic block, it is inferred from the block summary. Then, the symbol is looked up in the block summary of every previous basic block that is linked with a block edge to the current basic block. If the lookup in the block summary succeeds, the inferred symbol is fetched. The dimension of an `ArrayDimFetch` symbol is carried until a mapping `ArrayDimTree` symbol is found. The backwards-directed symbol lookup continues for each linked basic block and stops if a symbol of type `Value` is inferred or the beginning of the CFG is reached. At this point, all resolved symbols are converted to strings in order to perform context-sensitive string analysis [6]. The symbols `Value` and `Boolean` are converted to their representative string values. Data symbols of sources are mapped to a *Taint ID* (TID) that is used as string representation.

Next, each string is analyzed. The location of the TIDs within the markup is determined to precisely detect the context. For complex markup languages such as HTML

or SQL, a markup parser is used. With the help of the *sanitization tags* and encoding information of the linked data symbol, we check if the symbol is sanitized correctly according to its context. If a TID is found that belongs to an unsanitized source regarding the current context, a vulnerability report is generated. Unsanitized parameters or global variables are added to the function's summary as *sensitive parameter* or *global*. These are analyzed in the context of each function call.

3.3 Array Handling

By manually analyzing the code of the most popular PHP applications [28], we empirically found that a common way to write data into a database is by using arrays. An example is shown in Listing 5. In line 9 and 10, the array's *key* defines the table's column and the array's *value* stores the data to write. The separated array values are joined to a string again by using the built-in function `implode()` (lines 2/3). Based on this observation, we redesigned the handling of arrays by adding new data symbols. As a side effect, the handling of fetched database results in form of an array and the handling of the superglobal `$_SESSION` array is significantly improved.

```
1  function insert($table, $array) {
2     $fields = implode(",", array_keys($array));
3     $values = implode("','", $array);
4     mysql_query("INSERT INTO {$table}
5                  (".$fields.") VALUES ('".$values."')");
6  }
7
8  $new_user = array(
9     "name" => addslashes($_POST['name']),
10    "pass" => md5($_POST['pass']),
11 );
12 insert("users", $new_user);
13 // INSERT INTO users (name,pass) VALUES ('X', '123abc...')
```

Listing 5: Using arrays to write data to a database.

We model the popular built-in function `implode()` by adding the data symbol `ArrayJoin`. With the help of this symbol, it is possible to keep track of the *delimiter* that is used to join strings. If the symbol is inferred to an `ArrayDimTree` symbol, a `ValueConcat` symbol is created that joins all symbols of the `ArrayDimTree` symbol with the stored delimiter symbol.

Furthermore, we introduce the new symbol `ArrayKey`. It is used when the *key* of an array is explicitly accessed, such as in the loop `foreach($array as $key => $value)`. It is handled similar to the `Variable` symbol and is associated with the array's name. If the `ArrayKey` symbol is inferred into an `ArrayDimTree` symbol during data flow or taint analysis, a `Multiple` symbol containing all edges' symbols is returned. Built-in functions, such as `array_keys()` and `array_search()`, return all or parts of the available keys in an array and can be modeled more precisely with the `ArrayKey` symbol.

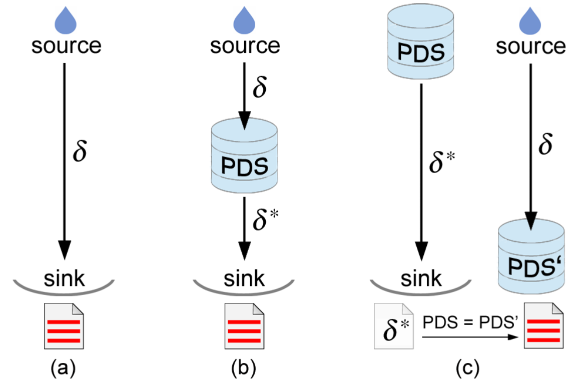

Figure 1: Data flow model of a conventional (a) and a second-order (b, c) vulnerability.

3.4 PDS-centric Taint Analysis

We now introduce our novel approach to detect second-order vulnerabilities. The data flow is illustrated in Figure 1 (b). Contrarily to a conventional taint-style vulnerability as shown in Figure 1 (a), a source flows into a PDS before it flows from the PDS into a sensitive sink. We model the data that is read from a PDS by new data symbols δ^* that hold information about their origin.

During code analysis, *taintable* PDS are identified. They are stored together with the minimum set of applied sanitization and encoding tags of the tainting data symbol δ. If one of the new data symbols δ^* is encountered unsanitized during the taint analysis of a sensitive sink, a vulnerability report is created if its originating PDS was identified as taintable.

If the PDS is not known as taintable, a *temporary* vulnerability report is created, as shown in Figure 1 (c). The report is connected to the data symbol δ^*. At the end of the code analysis, we decide if the data symbol originates from a taintable PDS by comparing its origin to all collected taintable PDS.

In the following, we introduce the analysis of writings to different PDS. Furthermore, our new data symbols δ^* are introduced that model the reading and access of data that is stored in PDS.

3.4.1 Databases

Modeling the data flow through databases is a complex task, mainly due to the large API that is available for databases and the usage of a query language. First, our prototype tries to obtain as much knowledge of the SQL schema as possible. Then we try to reconstruct all SQL queries during SQL injection analysis of 110 built-in query functions. Finally, the type of operation is determined, as well as the targeted table and column names. The access of data is modeled by new data symbols.

Preparation During the initialization of our tool, we collect all files with a *.sql* extension. All available CREATE TABLE instructions within these files are parsed so that we can reconstruct the database schema, including all table and column names as well as column types and length. If no schema file is found, each PHP file in the project is searched via regular expression. The knowledge of the database schema improves precision when data is read in an *unspecified* way, or when data is sanitized by the column type or length.

Writing A write operation to a database is detected if the SQL parser identifies an INSERT, UPDATE, or REPLACE statement. By tokenizing the SQL query, we determine the targeted table's name, all specified column names, and their corresponding input values. In case of an *unspecified write*, the parser makes use of the database schema. If an input value of a column contains a TID (see Section 3.2), the affected column and table name is marked as *taintable* together with the linked source symbol and its sanitization tags.

Reading If the SQL parser encounters a SELECT statement, we try to determine all selected column and table names. Multiple table names can occur if tables are *joined* or *unioned*. Alias names within the query are mapped and resolved. In case of uncertainty, the parser makes use of the database schema. Finally, a new ResourceDB symbol is mapped to the analyzed query function as return value. This symbol holds information about all selected column names in a numerical hash map and its corresponding table names.

Access In PHP, database result resources are transformed into arrays by built-in fetch functions (refer to Listing 2). We ignore the mode of access and let 89 configured fetch functions return a Variable symbol with the name of the resource. When an ArrayDimFetch symbol accesses the result of these fetch functions, it is inferred to the corresponding ResourceDB symbol. In this case, the carried dimension of the ArrayDimFetch symbol is evaluated against the available column names in the ResourceDB symbol. If the *asterisk* character is contained in the column list and the dimension is numerical, the database schema is used to find the correct column name. Otherwise, if the dimension equals a column name in the field list, a new DataDB symbol is returned that states which column of which table is accessed.

Sanitization Certain implicit sanitization is considered when dealing with SQL. If a column is compared to a static value within a WHERE clause in a SELECT statement, the return value for this column is sanitized. In this case, the static value is saved within the ResourceDB symbol and mapped to the column as return value. Furthermore, a sanitization tag for the used quote type is removed when data is updated or inserted to the database because one level of escaping is lost during writing.

3.4.2 Session Keys

The analysis of session variables does not require a complex markup parser or new data symbol. Instead, session data is handled similar to other global arrays. Taintable session keys are stored during the analysis phase.

Writing If data is assigned to a Variable or ArrayDimFetch symbol during block simulation and the symbol's name is $_SESSION, the assigned data is analyzed via taint analysis. If the assigned data is tainted, its resolved source symbol is stored into an ArrayDimTree symbol in the environment, together with the dimension of the $_SESSION symbol. This way, an ArrayDimTree is built with all taintable dimensions of the session array that link to the tainted source symbols and their corresponding sanitization tags.

Reading The access to session data is modeled by ArrayDimFetch symbols with the name $_SESSION and requires no modification. During taint analysis inside a user-defined function, session variables are handled as global variables. They are added to the function summary and they are inspected for each function call in a context-sensitive way. This avoids premature decisions about the taint status inside a function if the session key is overwritten before the function is called. Just as for a DataDB symbol, a *temporary* vulnerability report is created if a $_SESSION variable taints a sensitive sink.

3.4.3 File Names

To detect taintable file names, we collect file paths a user can write to. For this purpose, new data symbols model directory resources and their accesses. Whenever a path is reconstructed only partially, we use the same approach as in file inclusion analysis. Here, a regular expression is created and mapped to all available paths that were detected when loading the application files.

Writing To detect a file name manipulation with user input, we analyze 27 built-in functions such as copy(), rename(), and file_put_contents(). Additionally, file uploads with move_uploaded_file() are analyzed. Note that at the same time these built-in functions are sensitive sinks and generate vulnerability reports such as an *arbitrary file upload* vulnerability. The path argument is analyzed by conventional context-sensitive string analysis. If the path is tainted, we store it with its prefix as taintable. When no prefix is present, the file path of the currently analyzed file is taken. Additionally, if the source is not sanitized against *path traversal* attacks, all paths are assumed as taintable and a flag is set during analysis accordingly.

Reading We handle three different ways of opening a directory with PHP's built-in functions. First, we model the built-in function scandir() that returns an array, listing all files and directories within a specified path.

Second, we model the built-in function `glob()` that also returns an array that lists all files and directories specified by a pattern. We transform the pattern into a regular expression by substituting the pattern characters `*` and `?` into regular expression equivalents. Third, we model the built-in function `opendir()` which returns a *directory handle*. For all mentioned built-in functions, we reconstruct the opened path by string analysis and return a `ResourceDir` symbol that stores the path's name.

Access The returned result of `scandir()` and `glob()` is accessed by an array key. Since we do not know neither the amount nor the order of files in a directory, we return a `DataPath` symbol whenever a `ResourceDir` symbol is inferred from an `ArrayDimFetch` symbol, regardless of its dimension. For this purpose, we let the built-in function `readdir()` that is supposed to read an entry of a *directory handle* return an `ArrayDimFetch` symbol with an arbitrary dimension and the name of the directory handle. It is inferred to a `DataPath` symbol when the trace of the `ArrayDimFetch` symbol results in a `ResourceDir` symbol.

Sanitization In order to model sanitization that checks if a given string is a valid file name, 11 built-in functions such as `file_exists` and `is_file()` are simulated. We modified the sanitization check in a way that these functions only sanitize if there is no taintable file path found. For this purpose, a flag is set during taint analysis if sanitization of a source by file name is detected. The flag issues only a temporary vulnerability report that is revised at the end of the analysis regarding the ability to taint a file path.

3.4.4 Multi-Step Exploits

In order to detect multi-step exploits, we store all table names of all writing SQL queries that are affected by SQLi. Furthermore, we set a flag during the analysis process if an *arbitrary file write* or *arbitrary file rename* vulnerability is detected. At the end of the analysis, when the taint decision is made for data that comes from a PDS, multi-step exploit reports are added to the initial vulnerability. This is done for all vulnerabilities that rely on a `DataDB` symbol that is not tainted through second-order but which table name is affected by SQLi. Also, a multi-step exploit is reported if a `DataDir` symbol occurs and the flag for a *file rename* vulnerability was set. All session data is treated as tainted if an *arbitrary file write* vulnerability was detected. Additionally, any *local file inclusion* vulnerability is extended to a *remote code execution* if a file write or upload feature is detected.

Moreover, a SQLi vulnerability within a `SELECT` query returns a `DataDB` symbol with a *taint* flag. This flag indicates that all accessed columns are taintable by modifying the `SELECT` query during an attack. Thus, all columns of the `DataDB` symbol are taintable.

3.5 Inter-procedural PDS Analysis

We optimized the inter-procedural analysis to refine our string analysis results. Function summaries offer a high performance but they are also inflexible for functions with dynamic behavior. Thus, they can weaken the static reconstruction of dynamically created strings.

3.5.1 Multiple Parameter Trace

As we illustrated in Listing 5, modern applications often define wrapper functions for PDS access where more than one parameter is used within one sensitive sink. In this case, the approach of storing each parameter together with its prefixed and postfixed markup, and the corresponding vulnerability type as *sensitive parameter* in the function summary, is error-prone. When a call to this function occurs, the approach swaps the parameter symbol with the argument of the function call and traces it for user input. While this approach works fine for vulnerability detection, it leads to imprecision when it comes to string reconstruction. Because each argument is traced separately but both are used in the same sink, the result of one trace is missing in the result of the other trace. In Listing 5, for example, the table name is missing in the reconstructed query while the data is reconstructed from the `$new_user` array.

To circumvent this problem, we refined this approach for sinks that execute SQL queries or open file paths within a user-defined function. If multiple parameters or global variables are involved, all symbols are combined to one `ValueConcat` symbol. Then this symbol is stored in the function summary and analyzed for each function call. This way, each parameter is traced within one analysis and all results are present at the same time.

3.5.2 Mapping Returned Resources

Working with function summaries is very efficient when it comes to performance because each function only requires a single analysis on the first call. For every other call, the function summary is reused. However, a user-defined function might return a resource that has different properties for each call. For example, a `SELECT` query that embeds the parameter of an user-defined function as the table name returns a different `ResourceDB` symbol for every call, depending on the function's argument. If the resource is returned by the user-defined function, its symbol's properties change for every different function call.

As a solution, we add empty `ResourceDB` symbols to the function summary's set of return values for user-defined functions with dynamic SQL queries. Once the sensitive parameters are analyzed and the queries are reconstructed, a copy of these symbols is updated with the table and column information and used as returned data.

4 Evaluation

For evaluating our approach, we selected six real-world web applications. We chose the conference management systems *OpenConf 5.30* and *HotCRP 2.61* for their popularity in the academic field and *osCommerce 2.3.3.4* for its large size. Furthermore, we evaluated the follow-up versions of the most prominent software used in related work [3, 11, 30, 31]: *NewsPro 1.1.5*, *MyBloggie 2.1.4*, and *Scarf 2007-02-27*.

A second-order vulnerability consists of two data flows: tainting the PDS and tainting the sensitive sink. We evaluated our prototype for both steps and present the *true positives* (TP) and *false positives* (FP) in this section. In addition, we discuss the root cause for *false negatives* (FN) and outline the limitations of our approach.

4.1 PDS Usage and Coverage

To obtain an overview of the usage of PDS in web applications, we manually evaluated the total amount of different memory locations. Note that these numbers do not reflect how often one memory location is used at runtime. Then, we evaluated the ability to taint these memory locations by an application's user and compared it to the detection rate of our prototype. A PDS is defined as *taintable* if it can contain at least one of the following characters submitted by an application user: `\<>'"`. In total, we manually identified 841 PDS of which 23% are taintable. Our prototype successfully detected 71% of the taintable PDS with a false discovery rate of 6%.

4.1.1 Databases

Our implementation successfully recovered the database schema for all tested applications during the initialization phase. For evaluation, we categorized all available columns in the application's database schema by declared data type. Only columns with a *string* type, such as `VARCHAR` or `TEXT`, are of interest because they can store tainted data. As shown in Table 1, we found that on average about half of the columns are not taintable due to numeric data types such as `INT` and `DATE`.

Table 1: Column types in selected applications.

Software	Tables	Columns	Num	String
osCommerce	50	331	193	138
HotCRP	29	217	142	75
OpenConf	18	129	48	81
NewsPro	8	43	18	25
Scarf	7	37	22	15
MyBloggie	4	24	10	14
Total	116	781	55%	45%

We then carefully fuzzed a local instance of each application manually with common attack payloads in order to determine which columns of type *string* are taintable. Furthermore, we observed which columns were reported by our prototype implementation as taintable when the schema is available and when not. The results are compared in Table 2. Among the columns with a string type, 53% are taintable. As a result, only 24% of all available columns are not sanitized by the application or the columns' data type.

Table 2: Taintable columns in selected applications.

		Schema		No schema	
Software	Taintable	TP	FP	TP	FP
osCommerce	63	55	4	55	37
HotCRP	43	27	1	27	3
OpenConf	47	16	1	16	4
NewsPro	12	12	0	12	0
Scarf	10	10	1	10	3
MyBloggie	9	9	0	9	0
Total	184	70%	5%	70%	27%

For the rather old and simple applications, all taintable columns were detected by our prototype. The modern and large applications often use loops to construct dynamic SQL queries where reconstruction is error-prone. Overall, we detected 70% of all taintable columns. When the database schema is known, 5% of our reports are FP. The root cause is path-sensitive sanitization of data that is written to the database—a sanitization that our current prototype is not able to detect yet. The false discovery rate is higher if the database schema of an application is not found. In this case, a static analysis tool cannot reason about data types within the database and may flag columns of numeric data type as taintable.

4.1.2 Sessions

To obtain a ground truth for our evaluation, we again manually assessed the applications' code for all accessed keys of the superglobal `$_SESSION` array. Dynamic keys were reconstructed and keys in multi-dimensional arrays were counted multiple times. Then, we manually examined which session keys are taintable by the application's user and compared this to the analysis result generated by our prototype implementation. As shown in Table 3, we found that only 12% of the 52 identified session keys are taintable within our selected applications.

Our prototype correctly detected all taintable session keys. One FP occurred because the sanitized email address of a user is written to the session after it is fetched from the database. This FP is based on the previously introduced FP in identifying taintable columns. A custom session management in *osCommerce* led to exclusion from our evaluation.

Table 3: Taintable session keys in selected applications.

Software	Keys	Taintable	TP	FP
HotCRP	29	2	2	0
OpenConf	14	2	1	0
NewsPro	2	1	1	0
Scarf	4	0	0	1
MyBloggie	3	1	1	0
Total	52	12%	83%	16%

4.1.3 File Names

To evaluate the features that allow an application's user to alter a file name, we again manually assessed each application for file upload, file creation, and file rename features and counted the different target paths to obtain a ground truth. Next, we counted the collected taintable path names reported by our prototype. The results are shown in Table 4.

Table 4: Taintable paths in selected applications.

Software	Paths	Taintable	TP	FP
osCommerce	2	2	2	0
HotCRP	1	0	0	0
OpenConf	1	0	0	1
NewsPro	1	0	0	0
Scarf	1	1	1	0
MyBloggie	2	2	2	0
Total	8	63%	100%	16%

We found at least one feature in each of the application's source code to create a new file. However, half of the applications sanitize the name of the file before creating it. Our prototype detected all taintable path names. One FP occurred for *OpenConf*, where uploaded files are sanitized in a path-sensitive way.

Interestingly, a file upload in *Scarf* is based on a second-order data flow. The name of the uploaded file is specified separately and stored as a configuration value in the database before it is read from the database again and the file is copied. Because no sanitization is applied, an administrator is able to copy any file to any location of the server's file system which leads to *remote code execution*. This critical vulnerability was missed in previous work that also used this application for evaluating their approach [3, 31].

4.2 Second-Order Vulnerabilities

We evaluated the ability of our prototype to detect second-order vulnerabilities. Reports of first-order vulnerabilities are ignored for now. Our prototype reported a total of 159 valid second-order vulnerabilities with a false discovery rate of 21% (see Table 5 for details). In summary, 97% of the valid reports are *persistent XSS* vulnerabilities where the payload is stored in the database. Five *persistent XSS* vulnerabilities are caused by session data or file names. This is closely related to the fact that 94% of all taintable PDS we identified are columns in database tables (see Section 4.1) and sensitive sinks such as `echo` are one of PHP's most prominent built-in features [10].

Table 5: Evaluation results for selected applications.

Software	Files	LOC	TP	FP	FN
osCommerce	570	66 381	97	29	6
HotCRP	74	40 339	1	1	0
OpenConf	121	20 404	16	4	0
NewsPro	23	5 077	7	1	0
Scarf	19	1 686	37	8	3
MyBloggie	58	9 485	1	0	0
Total	865	143 372	159	43	9
Average	144	23 895	79%	21%	

Our evaluation revealed that second-order vulnerabilities are highly critical. Next to *persistent XSS* and file vulnerabilities, we detected various *remote code execution* vulnerabilities in *osCommerce*, *OpenConf*, and *NewsPro*. In the following, we introduce two selected vulnerabilities to illustrate the complexity and severity of real-world second-order vulnerabilities. It is evident that these vulnerabilities could only be detected with our novel approach of analyzing second-order data flows.

4.2.1 Second-Order LFI to RCE in OpenConf

OpenConf is a well-known conference management software used by many (academic) conferences. Our prototype found a *second-order local file inclusion* vulnerability in the user-defined `printHeader` function that leads to *remote command execution*. The relevant parts of the affected file *include.php* is shown in Listing 6.

```
1  function printHeader($what, $function="0") {
2    require_once $GLOBALS['pfx'] .
3      $GLOBALS['OC_configAR']['OC_headerFile'];
4  }
5
6  $r = mysql_query("SELECT `setting`, `value`, `parse`
7                    FROM `" . OCC_TABLE_CONFIG . "`");
8  while ($l = mysql_fetch_assoc($r)) {
9    $OC_configAR[$l['setting']] = $l['value'];
10 }
11 printHeader();
```

Listing 6: Simplified *include.php* of *OpenConf*.

When looking at the code, it does not reveal any vulnerability. Whenever the code is included, settings are loaded from the database and the user-defined function `printHeader()` is called. This function includes a configured header file and prints some HTML.

```
1  function updateConfigSetting($setting, $value) {
2     $q = "UPDATE `" . OCC_TABLE_CONFIG . "`
3          SET `value`='" . safeSQLstr(trim($value)) . "'
4          WHERE `setting`='" . safeSQLstr($setting) . "'";
5     return(ocsql_query($q));
6  }
7
8  foreach (array_keys($_POST) as $p) {
9     if (preg_match("/^OC_[\w-]+$/", $p)) {
10       updateConfigSetting($p, $_POST[$p]);
11    }
12 }
```

Listing 7: Simplified code to change settings in *OpenConf*.

However, as shown in Listing 7, it is possible for a privileged chair user to change any configuration setting. The configuration page does not specify an input field to change the *headerFile* setting. Nonetheless, by adding the key *OC_headerFile* to a manipulated HTTP POST request, the setting is changed. The loop over the submitted keys of the `$_POST` array in Listing 7, line 8, as well as the loop over the `$OC_configAR` in Listing 6, line 9, shows once again how important it is to track the taint status of PHP's array keys precisely.

A chair member can now include any local file of the system to the output. Additionally, because the software allows to upload PDF files to the server, our prototype added a multi-step exploit report. Indeed, if a PDF file containing PHP code is uploaded to the server and the *headerFile* setting is pointed to that PDF, arbitrary PHP code is executed. Moreover, our tool reported a SQL injection vulnerability that is accessible to unprivileged users. This allows any visitor to extract the chair's password hash (salted SHA1) from the database.

4.2.2 Second-Order RCE in NewsPro

Utopia NewsPro is a blogging software and was used in previous work for evaluation [29–31]. Our prototype reported a *second-order code execution* vulnerability in the administrator interface. Here, a user is able to alter the template files of the blog. The simplified code is shown in Listing 8.

```
1  $tempid = (int)$_POST['tempid'];
2  $template = mysql_real_escape_string($_POST['template']);
3  $updateTemplate = mysql_query("UPDATE `unp_template`
4          SET template='$template' WHERE id='$tempid'");
```

Listing 8: Simplified code to change the template in *NewsPro*.

The template code is read from the database in various places of the source code with help of the user-defined function `unp_printTemplate()` (see Listing 9). First, this function writes the template's code to a cache array (line 6) and then returns it from this array again. The example demonstrates the importance of inter-procedural analysis and array handling.

```
1  function unp_printTemplate($template) {
2     global $templatecache, $DB;
3     $getTemplate = mysql_query("SELECT name,template
4       FROM `unp_template` WHERE name='$template' LIMIT 1");
5     while ($temp = mysql_fetch_array($getTemplate)) {
6        $templatecache[$template] = $temp['template'];
7     }
8     return addslashes($templatecache[$template]);
9  }
10 eval('$headlines_displaybit = "' .
11          unp_printTemplate('headlines_displaybit').'";');
```

Listing 9: Simplified *Remote Code Execution* vulnerability in *NewsPro*.

At the call-site, the fetched template is evaluated with PHP's `eval` operator that executes PHP code (line 10). The template's code is escaped (line 8), however, the double-quoted value of the evaluated variable `$headlines_displaybit` allows to execute arbitrary PHP code using curly syntax. By adding the code `{${system(id)}}` to a template, the system command *id* is executed. Note that related work missed to detect this vulnerability, which is also present in prior versions.

4.3 Multi-Step Exploits

Our prototype reported two *arbitrary file upload* vulnerabilities and 14 SQL injection vulnerabilities. Because these vulnerabilities affect a storage operation, the stored data can be manipulated during multi-step exploitation. Our prototype found 14 valid multi-step exploits and a single FP as shown in Table 6.

Table 6: Reported multi-step exploits in selected applications.

Software	File TP	SQLi TP	SQLi FP	Multi-Step TP	Multi-Step FP
osCommerce	1	3	0	3	0
HotCRP	0	1	7	0	1
OpenConf	0	4	1	1	0
NewsPro	0	6	0	9	0
Scarf	1	1	0	1	0
MyBloggie	0	5	0	0	0
Total	2	20	8	14	1
Average	100%	71%	29%	93%	7%

All detected multi-step exploits consist of two steps and no *third-order* vulnerabilities were detected within our selected applications. In the following, we examine two multi-step exploits in *osCommerce* that lead to *remote command execution* to illustrate that these vulnerabilities can only be detected with our novel approach of analyzing multi-step exploits.

4.3.1 Multi-Step RCE in osCommerce

OsCommerce is a popular e-commerce software. For one of three reported SQLi vulnerabilities in *osCommerce*, our prototype additionally reported a *multi-step remote code execution* exploit. The SQLi is located in the backup tool of the administrator interface and shown in Listing 10. Here, a SQL file is uploaded to restore a database backup. Since the name of the uploaded file is later used unsanitized in a SQL query, an attacker is able to insert any data into the `configuration` table by uploading a SQL file with a crafted name. This enables another, more severe vulnerability: the table `configuration` stores a `configuration_value` and a `configuration_title` for each setting. Furthermore, a `use_function` can be specified optionally to deploy the configuration's value.

```
1 $sql_file = new upload('sql_file');
2 $read_from = $sql_file->filename;
3 tep_db_query("insert into " . TABLE_CONFIGURATION .
4   " values (null, 'Last Database Restore', 'DB_RESTORE',
5   '" . $read_from . "', 'Last database restore file',
6   '6', '0', null, now(), '', '')");
```

Listing 10: Simplified code of the *backup.php* file in *osCommerce* shows a SQLi through a file name.

When the list of configuration values is loaded from the database, the function name specified in the `use_function` column is called with the `configuration_value` as argument (see Listing 11, line 5). An attacker can abuse the SQLi to insert an arbitrary PHP function's name, such as `system`, to the column `use_function` and insert an arbitrary argument, such as *id*, to the column `configuration_value`. When loading the configuration list, the specified function is fetched and called with the specified argument that executes the system command *id*.

```
1 $conf_query = tep_db_query("select configuration_id,
      configuration_title, configuration_value,
      use_function from " . TABLE_CONFIGURATION . " where
      configuration_group_id = '" . (int)$gID . "'");
2 while ($configuration = tep_db_fetch_array($conf_query)) {
3   if (tep_not_null($configuration['use_function'])) {
4     $use_function = $configuration['use_function'];
5     $cfgValue = call_user_func($use_function,
6                   $configuration['configuration_value']);
```

Listing 11: Simplified code of the *configuration.php* file in *osCommerce* demonstrates a *multi-step RCE*.

4.3.2 Sanitization Bypass in osCommerce

Another *multi-step RCE* exploit was reported in *osCommerce* that involves a sanitization bypass. The previously mentioned backup tool of the administrator interface allows to specify a local ZIP file that is unpacked via the system command *unzip*. Here, the target file name is specified as an argument in the command line if the specified file name exists on the file system. The simplified code is shown in Listing 12.

```
1 if (file_exists(DIR_FS_BACKUP . $HTTP_GET_VARS['file'])) {
2   $restore_file = DIR_FS_BACKUP . $HTTP_GET_VARS['file'];
3   exec(LOCAL_EXE_UNZIP . ' ' . $restore_file . ' -d ' .
        DIR_FS_BACKUP);
4 }
```

Listing 12: A dynamically constructed system command in *osCommerce* includes the name of an existing file.

An attacker can bypass this check by abusing one of the file upload functionalities in *osCommerce*. By uploading a file with the name *;id;.zip* and afterwards specifying this file as backup file, the command *id* is executed. The semicolons within the file name terminate the previous *unzip* command and introduce a new command.

4.4 False Positives

Our prototype generated 43 false second-order vulnerability reports, leading to a false discovery rate of 21% for our selected applications. All false positives are based on the fact that our prototype is not able to detect path-sensitive sanitization. Thus, *persistent XSS* was reported in *Scarf* and *HotCRP* that are based on email addresses stored in the database. Our prototype erroneously identified these columns as taintable (see Section 4.1.1). The same error applies to a paper format in *OpenConf* which leads to four false positives. A user-defined sanitization function using path-sensitive sanitization based on its argument lead to 29 false *persistent XSS* reports in *osCommerce*. A false multi-step exploit was reported in *HotCRP* caused by a false SQLi report. By performing a path-sensitive sanitization analysis, these false positives can be addressed in the future.

4.5 False Negatives

Evaluating false negatives is an error-prone task because the actual number of vulnerabilities is unknown. Furthermore, no CVE entries are public regarding second-order vulnerabilities in our selected applications. However, it is possible to test for false negatives that stem from insufficient detection of taintable PDS. By pre-configuring our implementation with the taintable PDS we identified manually, we can compare the amount of detected second-order vulnerabilities with the number of reports when PDS are analyzed automatically.

As a result, only six previously missed *persistent XSS* in *osCommerce* were reported. Additionally, another taintable session key in *OpenConf* was reported, although the key does not lead to a vulnerability. Furthermore, we manually inspected the source code of the applications and observed that our SQL parser needs improvement. Three false negatives occurred in *Scarf* because our parser does not handle SQL string functions such as `concat()`. More complex SQL instructions might lead to further false negatives but are used rarely.

4.6 Performance

We evaluated our prototype with the implementation of our approach to detect second-order vulnerabilities (+SO) and without it (-SO). Our testing environment was equipped with an Intel i7-2600 CPU with 3.4 GHz and 16 GB of memory. The amount of memory consumption (M, in megabytes), scan time (T, in seconds), and second-order vulnerability reports (R) for our selected applications are given in Table 7.

Table 7: Performance results for selected applications.

	-SO Analysis		+SO Analysis		
Software	M[mb]	T[s]	M[mb]	T[s]	R
osCommerce	834	134	846	213	129
HotCRP	752	186	775	345	3
OpenConf	528	33	523	47	21
NewsPro	50	1	50	3	17
Scarf	39	1	40	14	46
MyBloggie	87	7	87	11	1
Total	2290	362	2321	633	217
Average	382	60	387	106	36

While the memory consumption does not increase significantly by adding second-order analysis, the average scan time increases by 40%. Note, however, that this includes 217 processed vulnerability reports the prototype would have missed without the additional second-order analysis. Furthermore, we believe that a total scan time of less than 11 minutes for our selected applications is still reasonable.

5 Related Work

Web applications are widely used in the modern Web and as a result, security analysis of such applications has attracted a considerable amount of research. We now review related work in this area and discuss how our approach differs from previous approaches.

Dynamic Analysis There are many different dynamic approaches to perform a security analysis of a given web application. For example, Apollo [1] leverages symbolic and concrete execution techniques in combination with explicit-state model checking to perform persistent state analysis for session variables in PHP. Sekar proposes syntax- and taint-aware policies that can accurately detect and/or block most injection attacks [23]. However, such approaches are typically limited to simple types of taint-style vulnerabilities.

There are also dynamic approaches to detect second-order vulnerabilities. For example, McAllister et al. present a blackbox scanner capable of detecting *persistent XSS* [19]. Ardilla [14] aims at detecting both SQL injection and XSS vulnerabilities by generating sample inputs, symbolically tracking taint information through execution (including through database accesses), and automatically generating concrete exploits. The typical drawbacks of such dynamic approaches are the limited test coverage and the missing ability to crawl a given site "deep" enough. This insight is confirmed by Doupé et al., who tested eleven black-box dynamic vulnerability scanners and found that whole classes of vulnerabilities are not well-understood and cannot be detected by the state-of-the-art scanners [7].

Static Analysis We perform static analysis of PHP code and use the concept of *block summaries* as proposed by Xie and Aiken [30] and later on refined by Dahse and Holz [6]. Our analysis tool extends these ideas and we improved the modeling of the language. More precisely, we introduce more data symbols (e.g., to analyze array accesses in a more precise way) and enhance the analysis of built-in functions such that we can perform a taint analysis for persistent data stores. Furthermore, we optimized the inter-procedural analysis to refine our string analysis results. This enables us to analyze the *two* distinct data flows that lead to second-order vulnerabilities: (i) source to PDS and (ii) PDS to sink. As a result, we are able to detect vulnerabilities missed by these approaches. Pixy [11] and Saner [2] are other static code analysis tools for web applications, but both do not recognize second-order vulnerabilities.

There are static analysis approaches that target other classes of security vulnerabilities. For example, Safer-PHP [25] attempts to find *semantic attacks* (e.g., denial of service attacks due to infinite loops caused by malicious inputs, or unauthorized database operations due to missing security checks) within web applications. RoleCast [24] identifies security-critical variables and applies role-specific variable consistency analysis to identify missing security checks, while Phantm [17] detects type errors in PHP code. Such kinds of software defects are out of scope for our analysis.

Static Second-Order Analysis The work closest related to our approach is MiMoSA [3]. It is an extension of Pixy [11] to detect multi-module data flow and work flow vulnerabilities. The data flow through databases is modeled, however, it uses a dynamic approach for the reconstruction of SQL queries. Moreover, it focuses on the detection of the work flow of an application and does not handle neither other types of PDS nor multi-step exploits. In comparison, only three data flow vulnerabilities were detected in *Scarf*, whereas our approach detected 37 second-order vulnerabilities and one multi-step exploit.

Zheng and Zhang proposed an approach to detect *atomicity violations* in web applications regarding external resources [31], which can be seen as being closely related to second-order vulnerabilities since such concurrency errors are a pre-condition for second-order exploits. They perform a context- and path-sensitive interprocedural static analysis to automatically detect atomicity violations on shared external resources. The tools *NewsPro* and *Scarf* are included into their evaluation, but the authors did not find any of the second-order vulnerabilities detected by our approach. As such, our approach outperformed prior work on static detection of second-order vulnerabilities.

6 Conclusion and Future Work

In this paper, we demonstrated that it is possible to statically model the data flow through persistent data stores by collecting all storage writings and readings. At the end of the analysis, we can determine if data read from a persistent store can be controlled by an attacker and if this leads to a security vulnerability. Our prototype implementation demonstrated that this is an overlooked problem in practice: we identified more than 150 vulnerabilities in six popular web applications and showed that prior work in this area did not detect these software defects. From a broader perspective, our approach can be broken down to the problem of statically reconstructing all strings that can be generated at runtime by the application and thus, is limited by the halting problem.

Future work includes modeling the data flow when prepared statements are used, supporting more SQL features, and analyzing data flow through file content. Also, path-sensitive sanitization and aliasing should be analyzed more precisely [32].

References

[1] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A., AND ERNST, M. D. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Softw. Eng. 36*, 4 (2010).

[2] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy* (2008).

[3] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-Module Vulnerability Analysis of Web-based Applications. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[4] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy* (2010).

[5] BOJINOV, H., BURSZTEIN, E., AND BONEH, D. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

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

[7] DOUPÉ, A., COVA, M., AND VIGNA, G. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2010).

[8] GUNDY, M. V., AND CHEN, H. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[9] HALFOND, W. G., VIEGAS, J., AND ORSO, A. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006).

[10] HILLS, M., KLINT, P., VINJU, J., AND HILLS, M. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)* (2013).

[11] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Static Analysis for Detecting Taint-style Vulnerabilities in Web Applications. *Journal of Computer Security 18*, 5 (08 2010).

[12] KERNIGHAN, B. W., AND PIKE, R. The Practice of Programming. In *Addison-Wesley, Inc* (1999).

[13] KHOURY, N., ZAVARSKY, P., LINDSKOG, D., AND RUHL, R. Testing and Assessing Web Vulnerability Scanners for Persistent SQL Injection Attacks. In *Proceedings of the First International Workshop on Security and Privacy Preserving in e-Societies* (2011), SeceS '11, pp. 12–18.

[14] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *International Conference on Software Engineering (ICSE)* (2009).

[15] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *ACM Symposium On Applied Computing (SAC)* (2006).

[16] KLEIN, A. Cross-Site Scripting Explained. *Sanctum White Paper* (2002).

[17] KNEUSS, E., SUTER, P., AND KUNCAK, V. Phantm: PHP Analyzer for Type Mismatch. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2010).

[18] LIVSHITS, B., AND CUI, W. Spectator: Detection and Containment of JavaScript Worms. In *USENIX Annual Technical Conference* (2008).

[19] MCALLISTER, S., KIRDA, E., AND KRUEGEL, C. Leveraging User Interactions for In-Depth Testing of Web Applications. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[20] MICROSOFT DEVELOPER NETWORK LIBRARY. Naming Files, Paths, and Namespaces. `http://msdn.microsoft.com/en-us/library/aa365247(VS.85)`, as of February 2014.

[21] NADJI, Y., SAXENA, P., AND SONG, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[22] SCHOLTE, T., ROBERTSON, W., BALZAROTTI, D., AND KIRDA, E. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *ACM Symposium On Applied Computing (SAC)* (2012).

[23] SEKAR, R. An Efficient Black-Box Technique for Defeating Web Application Attacks. In *Symposium on Network and Distributed System Security (NDSS)* (2009).

[24] SON, S., MCKINLEY, K. S., AND SHMATIKOV, V. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2011).

[25] SON, S., AND SHMATIKOV, V. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* (2011).

[26] SUN, F., XU, L., AND SU, Z. Client-side Detection of XSS Worms by Monitoring Payload Propagation. In *European Symposium on Research in Computer Security (ESORICS)* (2009).

[27] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRÜGEL, C., AND VIGNA, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2007).

[28] W3TECHS. World Wide Web Technology Surveys. `http://w3techs.com/`, as of February 2014.

[29] WASSERMAN, G., AND SU, Z. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)* (2008).

[30] XIE, Y., AND AIKEN, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium* (2006).

[31] ZHENG, Y., AND ZHANG, X. Static Detection of Resource Contention Problems in Server-side Scripts. In *International Conference on Software Engineering (ICSE)* (2012), pp. 584–594.

[32] ZHENG, Y., ZHANG, X., AND GANESH, V. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, pp. 114–124.