Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS

Eric Olsson
Chalmers University of Technology

Benjamin Eriksson
Chalmers University of Technology

Adam Doupé
Arizona State University

Andrei Sabelfeld
Chalmers University of Technology

Abstract

As web applications play an ever more important role in society, so does ensuring their security. A large threat to web application security is XSS vulnerabilities, and in particular, stored XSS. Due to the complexity of web applications and the difficulty of properly injecting XSS payloads into a web application, many of these vulnerabilities still evade current state-of-the-art scanners. We approach this problem from a new direction—by injecting XSS payloads directly into the database we can completely bypass the difficulty of injecting XSS payloads into a web application. We thus propose Spider-Scents, a novel method for grey-box database-aware scanning for stored XSS, that maps database values to the web application and automatically finds unprotected outputs. Spider-Scents reveals *code smells* that expose stored XSS vulnerabilities. We evaluate our approach on a set of 12 web applications and compare with three state-of-the-art black-box scanners. We demonstrate improvement of database coverage, ranging from 79% to 100% database coverage across the applications compared to the range of 2% to 60% for the other scanners. We systematize the relationship between unprotected outputs, vulnerabilities, and exploits in the context of stored XSS. We manually analyze unprotected outputs reported by Spider-Scents to determine their vulnerability and exploitability. In total, this method finds 85 stored XSS vulnerabilities, outperforming the union of state-of-the-art's 32.

1 Introduction

The web is a key enabler for today's ever-more digital world. Our society increasingly relies on web applications to support the financial, governmental, and military infrastructure. The dynamic functionality of web applications, coupled with the myriad of implementation technologies, makes developing a bug-free application challenging. Furthermore, these bugs can often manifest as security vulnerabilities. The complexity of modern systems and ever-powerful adversaries make securing web applications a *grand challenge*. Even the biggest web players such as Google and Meta still release vulnerable applications and services, which reflects in $12 and $2 million in bug bounties in 2022 respectively [15, 26].

Challenge of XSS. A particularly common class of web application security vulnerability is *Cross-Site Scripting (XSS)* [28], allowing attackers to inject JavaScript code into web pages. Astonishingly, XSS has persisted in the OWASP Top 10's list of most critical security risks to web applications for the past 20 years [48]. This remarkable persistence is reflected in bug bounties, with HackerOne reported paying over $4.7 million for XSS vulnerabilities in 2022 alone [17]. Securing web applications against XSS is difficult because there is no single general solution that prevents all XSS vulnerabilities [20]. Indeed, XSS vulnerabilities are context-dependent [42, 44], requiring that the correct output sanitization be used depending on the output context.

Stored XSS. Stored XSS, where the injection is stored and only later executed [28], is particularly challenging due to the disconnect that storage brings between the *source flow*, where the payload is input and stored, and the *sink flow*, where the retrieved payload is executed. Current vulnerability detection approaches [14, 32] have fundamental difficulties finding stored XSS.

Insufficiency of $*$-box approaches. The difficulty of securing a web application against XSS motivates the development of vulnerability detection tools [2, 7, 11, 33, 49, 54]. Web application vulnerability detection approaches can be classified as white-box, black-box, or grey-box based on what information is available (cf. Section 4):

White-box approach: White-box approaches [12, 19, 22, 24] usually statically analyze source code artifacts. Such static analysis is necessarily specific to the structure of the analyzed artifact, such as the server-side language or framework. Unfortunately, white-box vulnerability detection is fundamentally limited in its applicability to web applications because it is hard for white-box static analysis to precisely model the combined interplay of increasingly complex and dynamic client-side, database, and server-side behavior [43]. In addition, white-box analyses depend on the availability of artifacts,

further limiting their usage.

Black-box approach: More advantageously from a usability perspective, black-box vulnerability detection for web applications does not typically require access to source code and instead analyzes a running web application from the perspective of a user. Black-box scanners have been developed with various methods to better cover the increased attack surface of modern web applications [8], such as modeling server-side state [7], tracking data flows and fuzzing payloads [9, 10], modeling client-side state [25, 33], and combining multiple approaches [11]. However, while coverage of the attack surface has improved for some XSS, black-box scanners are often still unable to find even simpler stored XSS [32].

Grey-box approach: A common solution [1, 14, 18, 49, 50] is to combine black-box dynamic interactions of a running application, with white-box access. How these two information sources are combined varies. Sometimes, artifacts or non-standard interfaces only available with white-box access can be used to guide the otherwise black-box scan of a web application [47] . More commonly, a fully white-box static analysis of the application source code is combined with a black-box scan for dynamic runtime information [1, 14]. Combining these sources of information can mitigate some challenges inherent to otherwise using one approach in isolation. However, this combination still has a white-box component, from which its usability suffers.

While the previously described approaches constitute an exciting and active area of research, we identify a key consideration in the design space of $*$-box approaches. A core problem with finding stored XSS is that a *black-box scanner must find both the source and sink flows, and also understand the relationship between the two*, without any access to the web application's source code.

Approach. Inspired by recent work in improving binary fuzzing [34], our insight is to make stored XSS easier to find by relaxing the requirement that the scanner must find the source of a stored XSS. To do so, we supplement an otherwise black-box scanner with access to the database and allow the scanner to inject payloads directly into the database while scanning the running web application for sensitive sinks that output the inserted payload. This yields the benefit that a scanner no longer needs to find both source and sink nor understand their relation. The elegance of our approach is that it requires no knowledge of the web application source, only the database. Figure 1 illustrates the unique point our approach occupies in the design space of $*$-box approaches.

This unique position represents a paradigm shift from the domain of *application input* to that of *application state*, represented in the database. Challenges (presented in Section 3) stemming from fundamental problems with *-box approaches do not have to be solved by our method.

Based on these insights, we develop Spider-Scents, an approach to grey-box database-aware web scanning for stored

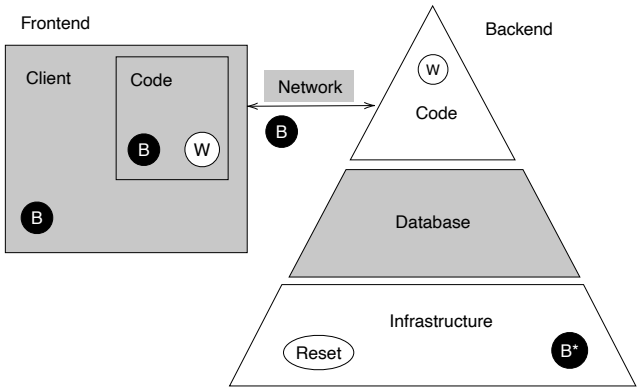

Figure 1: Access to components involved in a web application that different approaches need. Black-box approaches (B) have access to the front-end, and sometimes (B*) need infrastructure access to perform resets in the face of irreversible state changes. White-box approaches (W) have access to both front- and back-end code. Our approach, shaded in grey, accesses the front-end and database.

XSS. Spider-Scents injects payloads directly into the database and reports where database content is used in the HTML output without proper sanitization, flagging what we call *unprotected outputs*. This does not mean that all reports are stored XSS vulnerabilities, as the web application might be sanitizing the data on input. However, relying on input sanitization is against best practices for XSS prevention, as it is impossible to sanitize user input for every possible HTML output context. Indeed, the OWASP guidelines [31] postulate: "Apply Input Validation (using "allow list" approach) combined with Output Sanitizing+Escaping on user input/output," confirming that input validation alone is not enough. We term this result a *code smell*, an indication that something is wrong deeper within the application [13]. Even in the best case, where there is neither bug nor vulnerability, the application is fragile. Any new functionality added, such as creating a REST API, risks failing to properly sanitize user input. Therefore, even the unprotected outputs that are not currently exploitable stored XSS should be addressed by the web developer. Light manual analysis is required to verify a complete stored XSS vulnerability (see Section 6.7)

Non-vulnerable unprotected outputs constitute what we call a *dormant XSS*—they would be vulnerable, except that the *current* web application does not allow an exploit payload. The web application's evolution risks elevating a dormant XSS to a *complete XSS vulnerability*, even for security-wary applications. Our empirical study indeed confirms a dormant XSS vulnerability on WordPress, elevated to a complete vulnerability by a *real* published plugin (see Section 4.2).

Evaluation. We evaluate our approach across 12 web applications and compare our results with three state-of-the-art

black-box scanners. The applications range from reference applications used in prior work to latest versions of modern applications. Our results show that we cover (measured as how much of the database the scanner can change, described in Section 5.2.1) between 79% to 100% across all applications. In comparison, the other scanners cover 2% to 60% on average. We also find vulnerabilities that the other scanners are unable to detect. In total, we find 85 XSS vulnerabilities compared to 32 unique XSS for the other scanners. To further classify the impact of our findings we manually analyze the input protections and permission models of the Spider-Scents discovered vulnerabilities and determine that 59 are exploitable (and not self-XSS).

Contributions. We offer the following contributions:

- We present a novel approach for finding stored XSS vulnerabilities by injecting XSS payloads directly into the database, thus simplifying the detection of stored XSS. We present this in Section 4.
- We implement our approach into a prototype Spider-Scents, a semi-automated grey-box database-aware stored XSS scanner.
- We evaluate Spider-Scents and three state-of-the-art black-box scanners on 12 web applications. We present the results in Section 5 and analyze these in Section 6. Spider-Scents finds 85 XSS vulnerabilities across 7 applications.
- We systematize the relationship between *unprotected outputs*, *stored XSS vulnerabilities*, and *exploits* based on input protections and permissions models. Following this systematization, we manually analyze unprotected outputs reported by Spider-Scents to determine their vulnerability and exploitability. We also present this in Section 6.
- For the benefit of future research in this area, we share the source code of Spider-Scents[1].

Ethical considerations and coordinated disclosure. By actively scanning only our local clones of web applications in a controlled environment, we strictly avoid any harm caused by scanners on the web. We handle the discovered security vulnerabilities in accordance with the best practices of ethics in security [41]. We are in the process of reporting our findings to the affected vendors, following coordinated vulnerability disclosure for all discovered vulnerabilities. We report responses from vendors in Section 6.9.

2 Terminology

Here we attempt to systematize the terminology around XSS vulnerability analysis. Spider-Scents finds places in the web application where database content is used in the HTML output without proper sanitization. We term these *unprotected outputs*—output sinks where the output is not protected sufficiently against XSS. In contrast, there are also *protected outputs*: output sinks that are properly sanitized against XSS.

[1]Our implementation is available online at https://www.cse.chalmers.se/research/group/security/spider-scents/

We call a *complete XSS vulnerability* where user input flows to the unprotected output. Furthermore, that input must itself be an *unprotected input*—an input source lacking sufficient XSS protection. In contrast, there are also *protected inputs*: inputs protected with some combination of *sanitization* such as validation, stripping, or escaping. An unprotected output can also fail to be a complete XSS due to having *no input*.

If an unprotected input flows to an unprotected output, the web application has an *XSS vulnerability*. However, an XSS vulnerability is not necessarily *exploitable*, as this depends on the access control policy of the web application. The core question is if the user (or role) that injects the XSS payload can get the output on either another user or a role with greater permissions. We call XSS vulnerabilities that are not exploitable *self-XSS*, which have significantly less severity than exploitable XSS vulnerabilities.

3 Roadblocks for current XSS scanners

Automatically finding vulnerabilities in web applications remains a challenge despite active research in improving vulnerability detection. Stored XSS is especially difficult to find, as this type of vulnerability involves correctly injecting input into the application, where it will be stored by the database, and subsequently used in the web application's output incorrectly sanitized.

Black-box approaches can explore entire web applications without access or reliance on the underlying web application source code. It is possible for black-box scanners to track an entire stored XSS vulnerability from initial payload injection to vulnerable output. However, finding all such vulnerabilities for a black-box scanner is difficult, due to several challenges inherent to stored XSS that they must solve:

Vulnerable input validation. Web applications perform input validation (in client-side JavaScript and also server-side code) to ensure that the input data conforms to certain requirements. Web applications can check vulnerable inputs for validation that the scanner must pass and also inject an XSS payload into. As most black-box scanners use a pre-configured list of XSS payloads, it is difficult for them to create a custom XSS payload that also bypasses the vulnerable input validation.

Interdependent vulnerable input validation. Web applications validate all types of inputs. Often, a web application requires the user to fill out a set of inputs together. Consider a user registration form that might require the desired username, email address, zip code, and biography. All forms are required, the server-side input validation requires that the username be unique, the email address has a specific form, the zip code is five digits, and the biography has no validation and is vulnerable to stored XSS. Due to the *interdependence* of the four inputs, to find the stored XSS a black-box scanner must be able to provide a unique username (which is difficult for repeated injection attempts), correctly-formatted email address and zip code. As the number of inputs interdependent to vulnerable input increases, and as the input validation is

application-specific, it is more difficult for black-box scanners to generate the proper interdependent input to inject stored XSS payloads.

Vulnerable input modification. Web applications can also escape or modify user input. Consider a blog that accepts blog posts in markdown format that is transformed to HTML before storing them in the database. Because the black-box scanner has no knowledge of the server-side source code, it cannot know about this modification. If a vulnerable input is modified before being stored into the database, it is difficult for a scanner to create a custom XSS payload that can survive the modification, and, therefore, it is difficult to detect.

Multi-step vulnerable input. Web applications are stateful applications that can require a multi-step process before persisting user data. A classic example of this is the multi-step stored XSS in WackoPicko [8], where commenting on a picture requires first previewing the comment (which is output with sanitization) and then approving the comment (where it is output without sanitization). To our knowledge, the first black-box scanner that was able to automatically detect this vulnerability was Black Widow [11], published 11 years after WackoPicko was released. Therefore, vulnerable inputs that require multi-step interactions are difficult for black-box scanners to detect.

Vulnerable input identification. The core of stored XSS is that the XSS payload is stored in the database before being used as output. Even if a black-box scanner can correctly inject an XSS payload, it must be able to find where that input is output—otherwise, it will never detect the XSS.

4 Approach

Our goal is to overcome the challenges black-box scanners face (mentioned in Section 3) in detecting stored XSS vulnerabilities. Rather than take a completely black-box approach, we use a novel grey-box approach that includes *knowledge of the database* to help an otherwise black-box scanner find stored XSS vulnerabilities. Our idea is that by injecting the XSS payload directly into the database, and then scanning the web application for the payload's output, we can *completely bypass* several of the roadblocks black-box scanners face in detecting stored XSS: vulnerable input validation, interdependent vulnerable input validation, vulnerable input modification, and multi-step vulnerable input.

'Grey-box'. In web security, black-box has been synonymous with dynamic testing, as white-box is to static analysis. With this view, grey-box can appear to be defined as the combination of these two: dynamic testing and static analysis. Recent papers [14, 49] in grey-box web testing have supplied two definitions: the previous, based on *method*, or based on *access* or *visibility* to the application. Our approach is grey-box only by the second definition: we do not use static analysis, but we have access to the database.

Besides the application data contained, metadata such as table structure and associated relations are also assumed to be available, through the same database connection. In practical terms, SQL databases provide programmatic access to such metadata in the `INFORMATION_SCHEMA` tables [30].

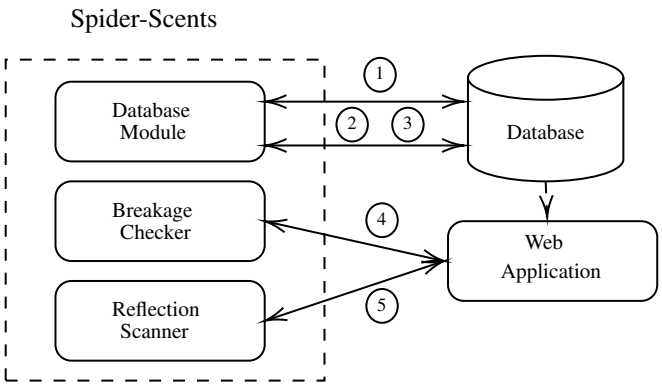

Figure 2: Overview of Spider-Scents' different components and their interactions with both the database and web application.

4.1 Overview

In contrast to existing state-of-the-art scanners, our database-aware method requires a paradigm shift from application *input* to *state*. Instead of solving problems associated with the input domain (such as crawling, modelling, or payload selection), we address new challenges of preparing the application state, selecting which part to modify, and analyzing its impact on the application.

Spider-Scents is our implementation of our novel grey-box scanning approach. Figure 2 shows a diagram of our approach. First ①, we prepare the web application for scanning. Next ②, we choose a database cell to modify, and ③ insert an XSS payload into the cell. We then validate ④ that the modified database did not break the application. Finally ⑤, we crawl the web application looking for reflections of the injected payload. This approach iterates as long as there remain untested database cells to modify.

4.1.1 Reports

Due to Spider-Scents injecting XSS payloads directly into the database, what it reports *is not XSS vulnerabilities*, but rather *unprotected outputs* (defined in Section 2). Therefore, the final step is to manually analyze the results of Spider-Scents. As our evaluation in Section 5 shows, many unprotected outputs are also XSS vulnerabilities—Spider-Scents finds 133 unprotected outputs in evaluated applications and 85 XSS vulnerabilities.

An interesting side-effect of finding so many vulnerabilities with Spider-Scents is that we realized that the *impact* of the discovered XSS vulnerabilities is critically important to contextualize the results. Specifically, we found that some of the XSS vulnerabilities were *self-XSS*, defined in Section 2

wherein the privilege required to store the XSS payload is the same as the user that views it. Of the 85 XSS vulnerabilities found by Spider-Scents, 26 are self-XSS while 59 are fully-exploitable.

4.2 Motivating Examples

Spider-Scents' database-aware scanning is particularly well suited to finding stored XSS vulnerabilities due to circumventing common challenges for black-box scanners. Here, we present examples of the types of issues that Spider-Scents is better at finding than previous work:

1. Fully-exploitable stored XSS that other scanners do not find.
2. Dormant vulnerabilities that become exploitable.
3. Self-XSS that other scanners do not find.

Fully-exploitable stored XSS. Even though they are capable of finding a fully-exploitable stored XSS vulnerability, other scanners often fail due to their inability to extensively explore the input surface of the web application, including satisfying vulnerable input and interdependent input validation. In Section 5, we further analyze the precise reasons why other scanners miss fully-exploitable stored XSS that Spider-Scents finds.

In addition, beyond needing a full XSS from input to output, the XSS vulnerability also must be exploitable. Determining the exploitability of an XSS currently requires manual analysis, as the exploitability of a vulnerability depends on application context such as the levels of user permissions within the application. Other presentations of scanners have generally not provided such analysis of their XSS results.

An example of a fully-exploitable stored XSS that other scanners fail to find is in the CMS Made Simple bookmarks functionality, which is shown in Figure 3. In this case, it is harder to find the input form, but easy to find the output from the database. Armed with this vulnerability, a user can perform XSS on an admin of CMSMS. Spider-Scents reports this as an unprotected output, and we manually confirm its exploitability.

Dormant XSS. Spider-Scents reports what it finds as unprotected outputs rather than stored XSS vulnerabilities, as it only finds the second half of a complete XSS workflow in unprotected outputs. Unprotected outputs do not always have an unprotected input flowing to them. However, we believe there is significant value in reporting unprotected outputs because they are a *code smell*—the web application has not followed the best practice of "always escape late" emphasized by both WordPress and WordPress VIP [51, 52]. While not a bug or vulnerability, an unprotected output is something that a developer should look at and fix.

In fact, we believe that an unprotected output in isolation can be considered a *dormant XSS*—it would be vulnerable, except that the current inputs for the web application do not allow an exploit payload due to either escaping, stripping, validation, no input possible, or some combination of these. The web application's evolution, either through future development or integration with other code (plugins, for example), can elevate a dormant XSS to a full XSS.

```
echo "<td><a href=\"editbookmark.php".$urlext."&amp;
    bookmark_id=".$onemark->bookmark_id."\">".$onemark->
    title."</a></td>\n";
echo "<td>".$onemark->url."</td>\n";
echo "<td><a href=\"editbookmark.php".$urlext."&amp;
    bookmark_id=".$onemark->bookmark_id."\">";
echo $themeObject->DisplayImage('icons/system/edit.gif',
    lang('edit'),'','','systemicon');
echo "</a></td>\n";
echo "<td><a href=\"deletebookmark.php".$urlext."&amp;
    bookmark_id=".$onemark->bookmark_id."\" onclick=\"
    return confirm('".cms_html_entity_decode(lang('
    deleteconfirm', $onemark->title) )."');\">";
echo $themeObject->DisplayImage('icons/system/delete.gif'
    , lang('delete'),'','','systemicon');
echo "</a></td>\n";
```

Figure 3: CMS Made Simple bookmarks functionality. A bookmark contains a title and a URL. Both the title and URL are correctly escaped in their respective first two cells in this snippet from CMSMS code, but the title is not protected in its inclusion in the delete button.

For example, Spider-Scents found that WordPress has unprotected output of both `display_name` and `user_nicename` from the `users` table. In the base WordPress application, there are no unprotected inputs to these columns—in fact, these are not modifiable after the creation of a user. However, *both* are exposed in a vulnerable version of the `username-changer` plugin [5] (code shown in Listing 4), and, therefore, when this vulnerable plugin is installed this unprotected output becomes a fully-exploitable XSS. [2] If WordPress followed their own best practices of "always escape late", this dormant XSS would not be possible (and Spider-Scents would not report it as an unprotected output).

It is also possible that some inputs to the database are kept constant. For example, in Hostel Management System, an application we evaluate in Section 5, a list of US states is hard-coded in the database. Spider-Scents finds unprotected outputs in this list. While not directly exploitable, these unprotected outputs could become a problem through extension of code, either in future versions of the application or with plugins. In our manual analysis, we quantify how many reports fall into this category and present them in the *NI* column in Table 2.

Self-XSS. Finally, an XSS vulnerability is not necessarily exploitable. An XSS can be unexploitable due to user permissions, such that the admin can only perform an XSS on themself. This self-XSS is the counterpoint to fully-exploitable XSS. An example of this is templates in MyBB. A template

[2] Vulnerable version of WordPress `username-changer` plugin https://github.com/evertiro/Username-Changer/blob/dd1976b05213d9895886da7f9a91515c52188344/includes/functions.php#L84

```
$qnn = $wpdb->prepare( "UPDATE␣$wpdb->users␣SET␣
    user_nicename␣=␣%s␣WHERE␣user_login␣=␣%s␣AND␣
    user_nicename␣=␣%s", $new_username, $new_username,
    $old_username );
$wpdb->query( $qnn );

$qdn = $wpdb->prepare( "UPDATE␣$wpdb->users␣SET␣
    display_name␣=␣%s␣WHERE␣user_login␣=␣%s␣AND␣
    display_name␣=␣%s", $new_username, $new_username,
    $old_username );
$wpdb->query( $qdn );
```

Figure 4: Username-Changer WordPress plugin vulnerable code: `display_name` and `user_nicename` are not sanitized.

Algorithm 1 Synthesizing data in the database.

```
rows ← 3
i ← 1
while i ≤ rows do
    row ← []
    j ← 0
    while j ≤ columns do
        append(row, increment(i))
        j ← j + 1
    end while
    insert(row)
    error ← breakage()
    if error then
        delete(row)
    end if
    i ← i + 1
end while
```

for the calendar functionality in this application can be modified to include executable JavaScript. However, only an admin user has the necessary permissions to add or modify this template. Spider-Scents finds this vulnerability, and we manually confirm that it is a self-XSS.

4.3 Preparing the web application

The initial step in our approach is to prepare the web application's database for scanning, as shown in step ① in Figure 2.

4.3.1 Database synthesis

Ideally, our method should scan an application with a *full* database. Steinhauser and Tůma note the importance of this as well [47]. However, they do not attempt to solve this and instead rely on somewhat complete configurations provided by the applications themselves, or other publicly-available manually-assembled data. For an e-commerce website, this means that the database already contains products, customers, and other data. To be able to discover an XSS-vulnerable workflow that spans multiple tables, data must exist in each table.

To address cases where data is lacking, more specifically empty database tables, we insert a constant number of rows of benign data matching the schema of the empty tables. We perform this simple algorithm shown in Section 4.3.1 on each empty table in the database.

columns is the number of columns in the schema for the empty table and *increment*(i) modifies a base value by the increment i. For example, integers have a base value of 0, dates have '1970-01-01 00:00:00', and strings have `a`. By modifying a constant value, we ensure that there are records correlated between tables by these deterministic values, to satisfy constraints common in web applications using normalized databases. We always insert 3 initial rows due to different auto-increments in the applications' database setups—we have observed empirically that these often start at 1, but not always.

While this naive solution is implemented and works well for our evaluation, the more general case of database synthesis is orthogonal to this work. Related work in this specific direction can be found in Section 7.5.

4.3.2 Reverting changes

We also periodically revert modifications done by Spider-Scents to the database, the rules for which are described in Appendix A.1. This is to add independence between our payload insertions, and also reduces our reliance on detecting application breakage, if we automatically revert based on other rules. Some rules we implement enforce independence across boundaries in the database; such as tables and columns.

To revert a database edit, from *newData* to *oldData*, perform:

```
error ← updateRow(oldData)
if error then
    removeRow(newData)
    insertRow(oldData)
end if
```

removeRow and *updateRow* can identify a row based on either keys or values. It is necessary to handle reverting changes in Spider-Scents, as built-in database functionality such as transactions cannot be open-ended, which is necessary to allow the simultaneous manipulation of both Spider-Scents and the web application backend of the database.

Reverting changes also happens when breakage is detected, conditions for which are covered in Section 4.6.

4.3.3 Logging in

Finally, we also must make sure that the web application itself is in a proper state to be scanned. Among other things, this means making sure that Spider-Scents is logged in. We automatically grab relevant details such as cookies, user agent, and the user URL (dashboard) at the start of each scan. Spider-Scents uses Selenium to interact with a headless Chrome instance with a custom extension to record this information, after having been supplied with the necessary credentials (username, password, and login page). These client details, which web applications often use to identify users, are then

Algorithm 2 Discovering sensitive rows.

```
i ← 1
while i ≤ rows do
    row ← retrieve(i)
    error ← delete(i)
    if error then
        sensitive(row)
    end if
    crawl()
    error ← insert(row)
    if error then
        sensitive(row)
    end if
    i ← i + 1
end while
```

re-used throughout the rest of the scan.

4.4 Choosing a database cell

For each cell in the database, Spider-Scents checks if the cell is suitable for our XSS payload. We ensure that the schema for the cell's column specifies that it is a text field with a length that can accommodate our payload, described in Section 4.5.

4.4.1 Avoiding sensitive rows

We also must consider if the web application is particularly *sensitive* to changing a specific cell's value. If the application tries to revert an injected database value, reset a table or row, or even insert a conflicting row, various problems can occur. This can either be due to the web application not having a robust recovery method that can handle Spider-Scents's admittedly unexpected meddling in their database tables, or due to Spider-Scents losing track of the state the database is in. In either case, we avoid such sensitive cells by first probing the database for their existence.

For each table in the database, we discover sensitive rows as shown in Algorithm 2, where *rows* is the number of rows in the table, $retrieve(i)$ and $delete(i)$ perform appropriate actions on the i^{th} row in the table, $insert(row)$ inserts the row into the table, $sensitive(row)$ marks that this row in the table is sensitive to changes, and $crawl()$ crawls the application to induce reverts, conflicts, or resets in the database.

4.4.2 Choosing a cell

How a table in a database is used by an application also matters, and, unfortunately, this usage is not always fully specified in the database schema.

If a table has uniform types of rows, then it might be sufficient to modify cells in a single row. However, this assumption only holds if, for instance, the access control policies of the application specify that all users can see the data in all rows.

This also assumes that each entry in the table is handled identically by the web application code, which might not be a correct assumption. Indeed, not all tables have only such uniformly typed entries; a counterpoint is tables that store key-value data, where the interpretation of a value cell depends on a key cell identifying it. Given the existence of such key-value tables and the fact that reflection code for entries can change based on the contents of a particular entry, we must iterate across all rows in a table.

In addition, the order of cell changes does impact which XSS-vulnerable workflows are discovered. We implement different traversal orders, primarily based on iterating tables in alphabetical order. From there, the scanner either chooses cells (that satisfy payload requirements and are not sensitive) based on the iteration of rows or columns.

4.5 Payload insertion

In contrast to traditional black-box approaches, Spider-Scents does not interact with the web application in an attempt to insert data from user input to a particular database cell. Instead, Spider-Scents inserts the payload directly into a database cell (step ③).

Once a suitable cell has been found, Spider-Scents generates a unique ID to use for the payload. The payload we use is inspired by payloads used in prior work [11]. This structure helps reduce false positives with dynamic XSS detection. We use the following template, where `ID` is replaced with a unique generated ID:

```
"'><script>xss(ID)</script>
```

While other payloads, such as a general-purpose `img` payload, can be more or less suitable depending on the context of the reflection, we consider the problem of creating smaller or context-specific XSS payloads to be orthogonal to this work. Using this payload can lead to false negatives, for example, if the reflection happens in a `textarea` or `title` tag.

4.5.1 Structured data

This is with the caveat that we *do consider* the presence of *structured data* in a cell. Spider-Scents' iterative modification of database cells, and searching for these individual reflections in the web page, has the implicit assumption that it is *individual* data stored in database cells in a one-to-one correspondence from web input to database cell. This might not be true—data might be split up into multiple cells or combined within a single cell. A payload would need to be either split across cells in the first case or multiple payloads combined in the second. Correlating changes across cells is considered out of scope for this work.

However, when data is combined within a single cell, we consider this as structured data. Spider-Scents modifies payloads to fit some common types of structured data. If a known structure is detected by parsing cell data, the payload will be delivered to each valid location for it. Currently, Spider-Scents implements a customized payload for PHP serialized objects and image paths. The support for these formats is chosen to demonstrate this approach and can lead to false negatives due to not handling more widely-used structured data formats,

Algorithm 3 Determining breakage.

```
i ← 1
broken ← 0
while i ≤ length(urls) do
    current ← request(urls[i])
    broken_url ← request(urls[i])
    if broken_url then
        broken ← broken + 1
    end if
    i ← i + 1
end while
if broken/length(urls) ≥ threshold then return broken
end if
```

such as JSON.

4.6 Application breakage

After inserting a payload, the changing database values can have catastrophic effects on the web application's functionality, which we call *breakage*.

While breakage can happen when using an application through its standard functionality, it is even more likely when Spider-Scents directly modifies the database and possibly inserts values that the application has not defensively coded against.

From a black-box scanner's perspective, such internal application-specification breaking changes are not available. However, similar behavior, when reachable on standard interfaces, is regarded as *irrecoverable state changes* [7]. While it may be intended functionality, such an irrecoverable state change overlaps with our notion of application breakage.

Guarding against breakage is inherently a tradeoff. On the one hand, changing the website's domain to an XSS payload in a CMS such as WordPress will rewrite all links, including admin ones, making the application unusable. On the other hand, new vulnerable behavior of the application might have been discovered instead.

Entirely black-box approaches will not be able to recover from such a breaking, irrecoverable state change. Some scanners add infrastructure access (shown in Figure 1) to be able to reset the application when this happens [7, 10]. With our position in the database, while we are more prone to cause breakage, we can also *identify and fix it*, without any additional access to the application.

Therefore, as noted by step ④ in Figure 2, Spider-Scents will dynamically scan the application looking for signs of breakage. If any are found we can reset the exact cell we changed that caused the breakage, even in the case when the web application is unusable.

Web application breakage across a web application is inferred by Algorithm 3, where *urls* is a list of URLs for distinct web pages in the application, $compare(baseline, current)$ compares the current status of a web page to a baseline measurement, and *threshold* is a threshold specified for how many pages can be acceptably broken in a web application.

$compare(baseline, current)$ can be implemented to compare measurements of a web page response based on HTTP status codes, linked content, length, or other heuristics. The approach we take is a combination of status codes and linked content.

4.7 Reflection scanning

Finally, we dynamically exercise the application with a reflection scanner (step ⑤). This scanner will crawl the application and report back on all the IDs that it finds. Here we differentiate between reflected JavaScript payloads that are executed, i.e. unescaped, and cases where we find the IDs in text. We do not try to find mangled or encoded payloads.

Spider-Scents uses *Black Widow* [11], the source code of which is available[3], with minor modifications to facilitate communication between modules, as its reflection scanner.

4.8 Manual analysis

From the Spider-Scents reports we manually analyze the unprotected outputs to determine if the payloads could be added from the web application. Once an input element is found we supply valid data and ensure the database is updated accordingly. Next, we add our payload and record if it is (1) rejected due to validation, (2) escaped, or (3) sanitized. We then repeat this for all inputs relating to the column.

5 Evaluation

We evaluate our approach by analyzing 12 different web applications and report on the number of stored XSS vulnerabilities found.

We compare Spider-Scents with a combination of up-to-date academic and open-source scanners that find stored XSS in Arachni [2], Black Widow [11], and OWASP ZAP [54].

5.1 Web applications

Similar to previous works [7, 11, 33, 49] we test both old applications and new modern ones. We divide the target applications into two sets. The five reference applications (that have known CVEs) and are used in prior work are: SCARF [45], Hospital Management System [35], User Registration & Login and User Management System [36], Doctor Appointment Management System [37], and Hostel Management System [38].

For modern, complex, applications we use these seven: CMS Made Simple [46], Joomla [21], MyBB [27], OpenCart [29], Piwigo [39], PrestaShop [40], and WordPress [53], Statistics that describe the applications chosen, their version numbers, and their usage in evaluation by prior work is provided in the Appendix Table 4.

[3] https://github.com/SecuringWeb/BlackWidow

Applications are largely chosen based on those evaluated by the authors of jäk [33], Black Widow [11], and Witcher [49][4], as these represent the current state-of-the-art in academic black-box web scanners. We restrict the evaluation to those that are database-backed.

Note that while we have selected applications based on the prior criteria, we choose the *latest version* of each application. Therefore, the unique vulnerabilities found by Spider-Scents are also *new*.

5.2 Experimental setup

In this section, we present the experimental setup used for the evaluation of Spider-Scents and comparison with other scanners.

5.2.1 Performance metrics

We focus our evaluation on three metrics: database coverage, vulnerabilities, and exploitability.

Database coverage. To successfully execute a stored XSS payload the scanner must first write the payload to the database [5]. By comparing a snapshot of the database before and after each scan we can approximately measure what effect each scanner has on the database.

These snapshots allow us to more precisely determine where the scanner fails in storing a payload, and where Spider-Scents can benefit from directly adding the payload to the database. In addition, we also classify the changes as either benign or XSS payloads. If an XSS payload is added, we investigate if the scanner can find it.

Vulnerabilities. To evaluate our method's capability to find vulnerabilities we also record the number of vulnerabilities reported by each scanner.

There is no clear method, neither in literature nor suggested by the scanners' implementations, of how to differentiate between two different XSS vulnerabilities. This means that for a given web application functionality, different scanners can generate different amounts of reported XSS vulnerabilities. For example, a vulnerable search bar included on every URL could generate a reported XSS vulnerability for each URL. To level the playing field and allow for a fair comparison we manually inspect each vulnerability and cluster them based on their related functionality. This clustering is justified as an application of *root cause analysis*, a process already well-established and valued in software bug reports [16].

Furthermore, we only compare stored XSS results from other scanners. Reports from compared scanners are manually confirmed to either be stored XSS or non-stored XSS (reflected or DOM). During evaluation, Arachni finds 4, Black Widow finds 2, and ZAP finds 4 [6] non-stored XSS. Our method is unable to find these, as it is specific to stored XSS.

Exploitability. Scanners search for XSS vulnerabilities by injecting data with XSS payloads into the application. Afterwards, they search for this data, either statically, or dynamically. However, due to permissions, this does not guarantee that an attacker can abuse the discovered XSS. For example, if only the super user can inject the payload, the vulnerability is not exploitable.

As modern applications can have complex user and group permissions with different associated application views, it is difficult for a scanner to automatically reason about the *exploitability* of these possible injections. In this paper, we manually verify and report on the exploitability of each reported vulnerability. We divide this step into two parts, i) input protections and ii) permissions.

For input protection, we ensure that it is possible to add the XSS payload from the application to the database. If it is not possible (protected), we further categorize the input protection for the reported vulnerability.

No Input, when there is no usable input field allowing for writing to the database. For example, input fields that are only available during installation or constants, such as US states.

Escaping, when the application changes the user input, to prevent it from being interpreted in some context, before adding it to the database. E.g. transforming `<` to `<` makes the symbol safe to be included in HTML context.

Stripping, when some data is removed ("stripped") from the input. E.g. removing `<script>` from the user input.

Validation, when the application refuses to add the user input to the database if it does not satisfy some format, such as containing illegal characters.

For permissions, we consider the vulnerability exploitable if a less privileged user can add a payload that is executed on a page that a user with more privileges can access. For example, if a normal user can book an appointment whose XSS payload is executed in the admin dashboard, then we would consider this exploitable. However, if only the admin could add the payload to such an appointment then it would be equivalent to self-XSS.

5.2.2 Scanner configuration

We configure other scanners to make as fair a comparison as possible. While we focus on stored XSS, web scanners can scan for a plethora of other vulnerabilities, including SQL injection, command injection, and local file inclusion.

For this evaluation, we configure each scanner to only focus on finding XSS vulnerabilities. Furthermore, to allow scanners a better chance to authenticate and stay authenticated, we make slight modifications to the *web application*. First,

[4] Witcher only supports detecting SQL Injection and Command Injection, therefore we do not compare against it.

[5] Assuming the payload is stored in the database; see Section 6.3 for an example of a stored XSS in the filesystem.

[6] In general, non-stored-XSS found by scanners is the difference between columns *R* and *S* in Table 1. However, due to the false positive reports by ZAP in the Hospital Management System (see Section 6.2), ZAP only finds 4 non-stored-XSS in the Doctor Apt. and Hostel applications.

we ensure the index page has a link to the admin login. Secondly, we rewrite the POST parameters server-side to match the correct user. This will level the playing field, as scanners prefer different authentication methods. Other scanners are configured to limit their runtime to 8 hours, similar to prior work [11, 49]

Configuration of parameters specific to Spider-Scents can be found in Appendix A.1.

5.3 Comparison results

In addition to the most direct comparison statistic—vulnerabilities found—we also collect a new statistic for this problem: *database coverage*. This is motivated by the different approach taken by Spider-Scents.

Database coverage. Modifying data in the database is required to detect stored XSS in database-backed applications. As such, we record the number of unique columns in the database each scanner modifies. In the case where an entire row is added, we give the scanner credit for all columns in the table. In our analysis in Section 6.4, we look more closely at the data inserted by the scanners.

In Appendix Table 5, we present the database coverage of each scanner and compare them to Spider-Scents. We further visualize this in Figure 5. As is evident, Spider-Scents can affect a much greater portion of the database compared to the other scanners. We cover between 79% to 100% while the other scanners cover between 2% and 60% on average. This shows that black-box scanners are still limited in how much they can affect the database, and subsequently, how well they can detect stored XSS.

There are cases where other scanners affect columns that Spider-Scents does not modify: For example, on CMSMS, both Arachni and ZAP affect columns that Spider-Scents does not. In this particular case, it is the `cms_adminlog.username` and the `cms_users.username` columns. Both these have a max length of 25 while our payload is 30 characters. We discuss these cases in more detail in Section 6.1.

Database application mappings. Our approach generates a mapping from database tables and columns where a payload is inserted, to the URLs where the payload is found. In Figure 6 we show such a mapping for the application Piwigo, where the red lines indicate unprotected output and the black lines indicate protected output.

XSS results. In this section, we compare the reported XSS vulnerabilities by each scanner. In Table 1, we present reports by each scanner in column *R*, manually confirmed stored XSS in column *S*, and manually verified and de-duplicated reported vulnerabilities in column *V*. Note that anything we find in column *V*, all black-box scanners should report as well.

Reports by other scanners are of XSS vulnerabilities. However, the Spider-Scents scanner reports unprotected reflections, not XSS vulnerabilities (see Section 4.1.1). Therefore, column *S* is inapplicable, and undefined for Spider-Scents.

Table 1: XSS vulnerabilities reported (and manually verified) by each scanner. *R* - All XSS or unprotected outputs reported by the scanner, *S* - Confirmed stored XSS, *V* - Verified and de-duplicated with our unique finds in parentheses.

Scanner	Arachni			Black Widow			ZAP			Spider-Scents		
	R	S	V	R	S	V	R	S	V	R	S	V
CMSMS	0	0	0	0	0	0	0	0	0	18	-	8 (8)
Doctor Apt.	5	2	2	1	0	0	4	1	1	8	-	4 (2)
Hospital	5	4	4	4	4	4	26	1	1	33	-	30 (22)
Hostel	13	13	13	3	3	3	0	0	0	23	-	19 (6)
Joomla	0	0	0	0	0	0	1	0	0	9	-	0
MyBB	0	0	0	0	0	0	0	0	0	6	-	6 (6)
OpenCart	0	0	0	0	0	0	0	0	0	6	-	0
Piwigo	0	0	0	1	1	1	0	0	0	5	-	1 (1)
PrestaShop	0	0	0	0	0	0	0	0	0	3	-	0
SCARF	0	0	0	10	9	9	0	0	0	12	-	11 (2)
User Login	0	0	0	0	0	0	0	0	0	3	-	3 (3)
WordPress	0	0	0	0	0	0	0	0	0	7	-	3 (3)
Total	23	19	19	19	17	17	31	2	2	133	-	85 (53)

Overall, our approach finds 85 stored XSS vulnerabilities that other scanners should be able to find, compared to the 15, on average, that they do find. 53 of these vulnerabilities found by our approach are unique and new.

With the exception of the Piwigo vulnerability Black Widow finds, which we discuss in Section 6.3, we find all stored XSS the other scanners find.

Notably, classic black-box scanners still struggle to find stored XSS, as indicated by the relatively low numbers in Table 1. There are some notable outliers, such as ZAP reporting 26 XSS on the Hospital Management System. However, as later clarified by manual analysis, this is a single stored payload being mislabeled as multiple XSS vulnerabilities.

Exploitability. While scanners report on user input being executed as JavaScript, they fall short of understanding the *exploitability* of the vulnerability. In this section, we break down the vulnerabilities we find with Spider-Scents into unprotected output, unprotected inputs, and unprotected permissions, defined in Section 2. We define the unprotected input as an input field where it is possible to add a payload without it being escaped, stripped, or subject to validation, as described in Section 5.2.1.

In Table 2 we present the results from our approach and exploitability analysis. Interestingly, we note that there is a diverse mix of input protection methods, even within one application. For example, in CMS Made Simple, escaping is used for the user's first and last name, while stripping is used for the email, and validation is used for the content alias. Nevertheless, the application still failed to properly sanitize its output.

Moreover, complex and dynamic user roles in modern applications make it difficult to automatically reason about the impact of XSS. For example, in CMSMS there is a binary

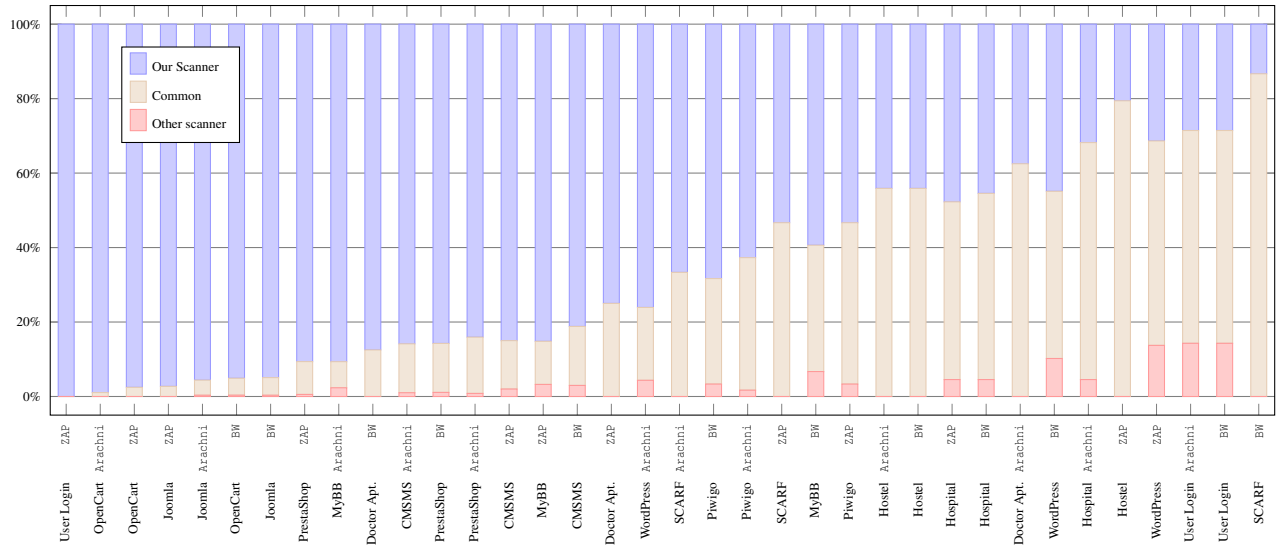

Figure 5: For every bar we present the fraction of database columns affected. First, on top, the fraction of columns only Spider-Scents finds, the middle shows the fraction of columns both scanners find, and finally, on the bottom, the fraction of columns only the other scanner finds.

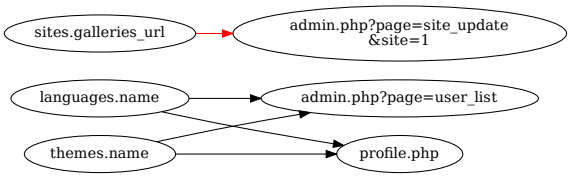

Figure 6: Subset of results from scanning the Piwigo web application. Black lines indicate protected outputs while red lines indicate unprotected outputs.

option for permission to modify bookmarks, that can be assigned to any user group. Any user with this permission can abuse an XSS to gain privileges. In contrast, MyBB has a strict separation of admin configurations and forum moderation configurations. This means that any XSS a scanner finds, including ours, while authenticated as an admin in MyBB, could be regarded as self-XSS as only trusted parties control the input. Similarly in Piwigo, the page title is vulnerable to XSS, however, only the admin can change it. As we see in Table 2, while both MyBB and CMSMS fail to escape all outputs, MyBB is less exploitable due to its stricter permissions.

In the Doctor Appointment Management System, neither output protection nor input protection is used. Despite this, two vulnerabilities are not exploitable because of permissions. Specifically, while users can change their own email address, only doctors (super users in this context) can change a doctor's name. Still, these are not fully protected, as the application with its default settings is also vulnerable to CSRF attacks. These vulnerabilities can be combined to exploit the XSS. Therefore, we believe it is useful for developers to learn where unprotected outputs are so that they can be fixed, even if they are protected by permissions.

6 Analysis

In this section, we investigate our results and highlight limitations of both black-box scanners and our approach.

6.1 Database coverage

While our method generally achieves higher database coverage, there are some interesting cases where other scanners still perform better in this metric.

In our evaluation, the portion of the database we miss and other scanners can reach is a result of our payload's length. Our payload length is always at least 30 characters long, making it too big for some cells. For example, on Piwigo Black Widow can affect the `oc_customer_ip.country` column, which only holds two characters. ZAP also modifies `mybb_templates.version`, with a size of 20 characters. This could be enough for some XSS payloads.

In theory, we can miss finding possible vulnerabilities if changing a non-text (numeric or date) value is necessary to trigger a vulnerability. Foreign key constraints can also cause problems but are less common in the text fields we focus on.

Table 2: Exploitability of reported vulnerabilities. *T* - Total reflections, *NI* - No Input, *VA* - Validation, *ST* - Stripping, *ESC* - Escaping, *P* - Permission, *EXP* - Exploitable. * A CSRF vulnerability could be abused to exploit it. ** Poor authentication validation allows privilege escalation.

	T	NI	VA	ST	ESC	P	EXP
CMSMS	18	3	1	3	3	0	8
Doctor Apt.	8	4	0	0	0	2*	2
Hospital	33	3	0	0	0	6*	24
Hostel	23	4	0	0	0	4**	15
Joomla	9	5	0	4	0	0	0
MyBB	6	0	0	0	0	6	0
OpenCart	6	0	0	0	6	0	0
Piwigo	5	1	3	0	0	1	0
PrestaShop	3	0	3	0	0	0	0
SCARF	12	1	0	0	0	7*	4
User Login	3	0	0	0	0	0	3
WordPress	7	0	0	1	3	0	3

6.2 False positives

Black-box scanners use a variety of methods to detect injected XSS payloads, which can result in false positives. ZAP, for example, incorrectly identified XSS in WordPress. It statically found the injected token `;alert(1);` in a JavaScript context. However, the token was inside a string, which in this case it is not possible to break out of.

Confusing multiple payloads is another problem many scanners face. In the Hospital Management System, ZAP can successfully inject an XSS payload into the database. However, it does not detect this as a stored XSS, and is confused when it later scans for DOM-based XSS with the same payload, `alert(5397)`, which ZAP does find. This is caused in part by the number 5397 not being random but a constant, defined in the code as `UNLIKELY_INT`. Therefore, for a single stored XSS vulnerability, ZAP instead reports 26 DOM-based XSS. In this case, mistaking DOM XSS for stored XSS can impact developers who are unable to reproduce the results when the database is reset.

Our approach can avoid many of these problems by using unique IDs for each cell in the database and dynamically testing that each payload is executed. As such, similarly to Black Widow, we have a low rate of this type of false positive.

In contrast to black-box scanners, our approach does not automatically verify that an unprotected input exists. Therefore, we might report a database cell that cannot be changed by the web application. For example, in the Hostel Management System, the list of US states were not escaped on output, but were all hard-coded. We argue that when more functionality is added, either through software updates or third-party code such as plugins, it can introduce a vulnerability, and as such these reports are important.

6.3 What we miss

In the complex setting of web applications, there might always be more unknown vulnerabilities. In the absence of ground truth, in line with previous work [9–11, 33, 47, 49], our false negative comparison baseline is the stored XSS results of other scanners.

In Piwigo our method can find a reflected value, but not XSS, for a value in the configuration table. However, upon further manual analysis, we note that the reason we did not find the XSS vulnerability was because of the payload chosen. In this case, the value was reflected inside a textarea, meaning a `</textarea>` tag was needed to break out and execute JavaScript.

Black Widow found one XSS on Piwigo that we missed. While we were able to add the payload and find the correct URL, the URL was too late in the reflection scanner's queue and was therefore never visited. Increasing the reflection scanner's timeout would solve this, at the cost of runtime performance.

We have only implemented a prototype that demonstrates the utility of our approach. As with all scanners, the choices taken in that implementation can lead to false negatives. False negatives can stem from missing SQL analysis that can limit our interaction with the database, such as foreign keys and other constraints, and triggers. Better authentication, and re-authentication, mechanisms would also improve our approach. Replaceable components to our method, such as the reflection-crawling and payload-selection modules, are shown to be the cause for some of the false negatives in this section.

In addition to our evaluated comparison with the other scanners, we also survey vulnerabilities from CVEs and previous academic papers to construct a dataset of known vulnerabilities in our choice of evaluated web applications. In this dataset, 33 previous reports covered 29 unique stored XSS vulnerabilities, of which we find 26 (details in Table 6). In WordPress, we miss a vulnerability that relies on an attacker-controlled server that returns a crafted message to a link embedded in a post. For all evaluated methods, including ours, this type of attack is out-of-scope. However, it should be noted that we do correctly mark the database column as unprotected output. In CMSMS, we miss a stored XSS vulnerability in the filename of uploaded files. Here the payload is not stored in the database but rather in the filesystem. Extending our method from databases to other storage mechanisms could be an avenue for future work. Finally, in OpenCart, we miss a stored XSS in the category description because we do not spend enough time crawling the application after database modifications, to scan for reflections of the inserted payload. After extending the reflection scanner's timeout, Spider-Scents was able to find this vulnerability as well.

6.4 What others miss

From the scanning results of the Doctor Appointment Management System, we can see that ZAP fails to detect multiple XSS vulnerabilities. By analyzing database snapshots before and after execution we note that ZAP was only able to insert data into the `tbldoctor` table. While it was able to add the string "ZAP" to the `FullName` column, it could not add an XSS payload. Furthermore, ZAP misses other tables, such as `tblpage` and `tblappointment`, that our method modifies and detects as unprotected output.

We also note cases where the compared scanners fail to find XSS due to a lack of database coverage. For example, in MyBB, no other scanner affects the vulnerable `mybb_usergroups.namestyle` column.

6.5 Exploitability

Both black-box scanners and our method will report on injected JavaScript being reflected and executed. However, as we see in Table 2, not all these executions could, in their current form, be exploitable. Interestingly, we note that web applications are relatively equally split on using sanitization and escaping on user input. Validation, on the other hand, is less common. Moreover, permissions also play an important role in protecting these *XSS vulnerabilities* from becoming exploitable.

6.6 Drop-in testing with Spider-Scents

In our evaluation, we assisted other scanners by modifying the web applications under test, so they could evade typical login checks. While beneficial for increasing their performance for the sake of comparison, such modifications are not ideal.

To demonstrate the applicability of Spider-Scents, we *do not modify* web applications when we run Spider-Scents. Therefore, we rely entirely on our breakage heuristics, automatic reverting, and automatic use of captured log-in details.

6.7 Manual analysis with Spider-Scents

Spider-Scents requires more manual effort to verify a vulnerability compared to a black-box solution. However, as Spider-Scents reports the corresponding database table and column, e.g. `users.email`, it is usually relatively easy to manually find relevant input fields and test for a working XSS payload, as described in Section 4.8. In our evaluation, it took an author approximately 15 minutes per report, on average. Preparing Spider-Scents to scan takes a similar time to other scanners, with the small addition of database credentials.

Further automation to ease analysis is possible, such as mapping input fields to the database, although this will require addressing general challenges of crawling, such as exploration and input validation [11].

6.8 Runtime performance of Spider-Scents

In contrast to other black-box scanners, which can run indefinitely [11], our method runs for a time proportional to the database of the application. In our evaluation, other black-box scanners are limited to a runtime of 8 hours. Spider-Scents almost always completed its scans within this time window, with the exception of modern applications Joomla and PrestaShop. Reference applications are scanned within minutes, while modern applications are scanned in hours.

Table 3: Runtime performance of Spider-Scents. Runtime is proportional to the size of the database of the application, reported in both the raw number of database cells and those that Spider-Scents can scan (satisfy payload requirements).

	Scan time	Database cells	Scannable
CMSMS	7:14	11844	4339
Doctor Apt.	0:08	195	87
Hospital	0:22	282	106
Hostel	0:13	205	90
Joomla	12:59	11584	4813
MyBB	4:21	15701	5321
OpenCart	1:39	31490	11553
Piwigo	1:07	1826	520
PrestaShop	32:29	44529	10745
SCARF	0:06	36	15
User Login	0:01	10	6
WordPress	1:55	385	868

We report the scan time of Spider-Scents in evaluation in Table 3. These times are collected on a laptop from 2021 with 8 cores and 16 gigabytes of RAM, running both the application's web server and the Spider-Scents scanner.

6.9 Coordinated disclosure

We have reported all new vulnerabilities to the affected vendors and will summarize their responses here. MyBB is planning to fix the vulnerabilities we reported in the upcoming 1.9 version. The CMSMS developers argue that any authenticated XSS (regardless of the specific user/group permissions) is not considered a vulnerability. Instead, they will revise their documentation to no longer motivate their permission model as a "security mechanism". WordPress, on the other hand, does consider some authenticated XSS as vulnerabilities, depending on permission. However, their security model differs from that evaluated. In our model, we considered any privilege escalation as a vulnerability, while WordPress developers consider *editor* and *admin* to be equivalent. As such, there does not seem to be a consensus among web developers as to how application permissions should be modelled. We are still waiting for a response from PHPGurukul for vulnerabilities in their multiple applications. However, these vulnerabilities have a clearer precedence with similar vulnerabilities to ours, e.g. *CVE-2023-27225*.

6.10 Summary

As the results show, Spider-Scents performs both better in database coverage and stored XSS vulnerability detection when compared to state-of-the-art scanners. Based on what vulnerabilities the other scanners miss and what we uniquely find, we believe the reason for this improved performance is because we bypass the majority of the roadblocks that current XSS scanners face (as defined in Section 3). Solving these challenges directly is a substantially harder problem [8], and will require solving fundamental challenges with crawling [11]. In many instances, the other scanners fail to get any data into the vulnerable database column for vulnerabilities only Spider-Scents finds, and in other cases when they do, only benign data is added. In general, the main problem current scanners face, which we bypass, is getting the payload into the database.

7 Related Work

7.1 Black-box scanners

Enemy of the State [7] models server-side state in different links and requests are identified that drive such state changes. Notably, this work recognizes the necessity of a solution to resetting a web application. In this case, the application is run in a VM, and the machine is reset to counteract irreversible state changes. Spider-Scents does not need a VM, and resets can be done in a granular and inexpensive fashion. Furthermore, Enemy of the State's access to the VM subsumes this paper's access to the database.

LigRE [9] and KameleonFuzz [10] also focus on server-side state. LigRE improves XSS detection with taint flow inference, and KameleonFuzz adds genetic algorithms for payload generation and modification. Similar to Enemy of the State, these approaches require the ability to reset the web application.

jäk [33] instead focuses on modelling client-side state. JavaScript APIs are hooked to be able to model dynamic behaviour. The crawler generates a navigation graph including this information.

CrawlJax [25] also models client-side state. Interactable candidate elements, such as clickable ones, are interacted with to extend the crawler's reach. A state-flow graph models the user interface.

Black Widow [11] identifies key fundamental challenges for black-box scanning. They mitigate them by combining navigation graphs, workflows, and inter-state dependencies in one XSS scanner. In contrast to prior work, Black Widow does not assume the ability to reset the web application.

7.2 White-box scanners

Saner [4] focuses on identifying improper sanitization to find vulnerabilities such as XSS and SQLi. Saner is limited to analyzing PHP, and even more to custom sanitization routines. To reduce complexities with application state – such as the database – Saner does not interact with a live instance of the web application, instead choosing to build a model of the sanitization process from static analysis results.

Restler [3] does not use the entire application's codebase, but instead only the REST API specification. Static analysis of this specification identifies inter-request dependencies to generate tests, which generate dynamic feedback execution to guide further testing. Similar to Spider-Scents in both analyzing a different artifact/interface than typical static or 'grey-box' analyses, Restler also focuses on bugs. Indeed, they note that vulnerabilities in a REST specification are unclear.

Sentinel [24] seeks to limit access to sensitive data in the database to SQLi attacks. The authors model web applications to identify invariants for the 'normal' functionality, which they use to examine queries and responses to block malicious SQL usage.

7.3 Grey-box scanners

Most prior grey-box approaches inform a white-box scan with some runtime information from a black-box scan, to reduce the false positive rate and generate a full exploit proof. We argue that only needing database access is more general than source code [7]. It is easier to apply our approach to a different web application. Being almost black-box, we are agnostic to the coding language and framework for the application's implementation. White-box approaches might not be able to handle obfuscated code. Obfuscated code can also be present due to extensions, such as plugins. We also do not replace the database or insert some proxy between the database and the application. This makes it easy to adapt our approach to other storage mechanisms.

webfuzz [50] instruments code for coverage, and uses this feedback to fuzz requests for detecting reflected and stored XSS. This approach is expensive - WordPress reaches 27% coverage in 2000 minutes.

Witcher [49] identifies issues with using grey-box coverage to guide a web application fuzzer for vulnerability discovery. A Fault Escalator is defined to detect when the application is in some vulnerable state, and guide the fuzzer to escalate that to a vulnerability. Together with a refined notion of coverage, Witcher can fuzz URLs to find command and SQL injection. This approach is limited to first-order reflected vulnerabilities and does not model application state.

Gelato [18] detects reflected and DOM-based XSS. Taint analysis is used to target exploration of the large state space of modern JavaScript.

Backrest [14] statically infers a model REST API, then uses coverage and taint feedback to drive fuzzing of requests for detecting SQLi, XSS, and command injection. While the motivations are to both improve coverage and runtime, the runtime improvements are more evident. Notably, XSS detec-

[7] Static analysis can also be performed on compiled binaries or intermediate representations. However, the same arguments against generality apply to those other artifacts.

tion, especially stored XSS, is reduced when provided with feedback. The authors point out the problem of following taint across the interface with storage in a database.

Chainsaw [1] implements automated exploit generation, where potential vulnerabilities derived from static analysis are dynamically tested, with successful executions being concrete exploits. Symbolic execution of PHP is used to find sources and sinks, as well as sanitizations/transformations along paths. The database is regarded as an additional input to the application, with the database schema consumed. Workflow-based vulnerabilities, such as stored XSS, are found within a comparable 600 minutes.

7.4 Database-aware grey-box web scanning

Similarly to Spider-Scents, Steinhauser and Tůma utilize a grey-box approach for detecting context-sensitive XSS using the database alongside a normal black-box scanner [47]. They deal with context-sensitivity in line with Context-Auditor [23].

Steinhauser and Tůma' s grey-box approach intercepts database and web application communication, injecting non-XSS payloads into the application by replacing data coming from the database.

XSS flaws are detected by black-box parsing of HTML responses from the application matching portions of payloads in responses, to detect the payload if the application applies some common encodings. The parser continues with the possible XSS flaw, and payloads are iteratively modified to avoid context encoding. Some automatic reports of XSS flaws must then be manually analyzed to identify vulnerabilities

This approach is substantially different from that of Spider-Scents. We achieve a different, more complete form of coverage, by iterating through the contents of the database, instead of only modifying values as they are retrieved by the requests from a black-box scanner. This approach ① skips missing database entries, with the database only populated by pre-provided configs or manually sampled data from public demo instances. We also scale differently; with the database modelled as *additional inputs*, this approach of extending HTTP request fields can become ② several *orders of magnitude slower* than the base black-box scanner. For efficiency, *all database injections* are combined per request, ③ which authors note increases breakage, without proposing a solution. The applications under test are also ④ substantially modified to aid the scanner. Finally, Steinhauser and Tůma's approach is implemented by ⑤ extending MariaDB, which replaces the database in the tested web application.

In contrast, Spider-Scents ① augments the database, ② is lightweight in our evaluation, ③ identifies and fixes application breakage, ④ works without modifying web applications, and ⑤ is implemented without a heavy-weight database replacement or proxy.

In terms of results, we share Joomla and PrestaShop. Steinhauser and Tůma verified 5 and 12 vulnerabilities in Joomla and PrestaShop, with their reports resulting in all Joomla and some PrestaShop flaws being fixed. As all Joomla flaws reported were fixed, the 9 we identify either are missed by their approach or come from further development of Joomla. Unfortunately, source code artifacts for this paper are unavailable, so we cannot do a direct comparison.

7.5 Database synthesis

SynthDB [6] is a recent work addressing the tangential problem of preparing database-backed web applications for security testing, such as vulnerability scanning, by synthesizing a database. In contrast to our simplistic approach, with the singular goal of having some data in every table while correlating inserted fields across tables, SynthDB uses concolic execution of PHP source to collect database constraints. These constraints are solved to uncover more program paths, while not violating 'database integrity'. Similar to Spider-Scents, the performance of scanners such as Burp is improved with this white-box preprocessing step.

Our approach also uncovers a separate problem - finding a *minimal* database. In the osCommerce application, there are over 3 million cells in the base application.Our approach must have its parameters tuned to handle this volume of cells. However, we have found this application's scale abnormal by several orders of magnitude; other web applications typically only have hundreds to tens of thousands of cells.

8 Conclusion

Black-box vulnerability scanners are the best tools currently available for democratizing security testing—allowing web developers with no security background or knowledge to proactively find vulnerabilities in their web applications. However, the twisted designs and logic of web applications make it difficult for black-box vulnerability scanners to even inject XSS payloads into the web application. Our approach cuts this Gordian Knot of properly supplying inputs to a web application—by injecting the XSS payloads directly into the database. We believe that this approach represents a step forward in automatic stored XSS detection, and the evaluation results show that our Spider-Scents prototype surpasses state-of-the-art black-box vulnerability scanners, while our manual systematization provides the necessary contextualization of vulnerability and exploitability to these results.

Acknowledgements This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), and the Swedish Research Council (VR).

References

[1] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *CCS*, 2016.

[2] Arachni. https://www.arachni-scanner.com.

[3] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE*, 2019.

[4] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S&P*, 2008.

[5] Username Changer. https://wordpress.org/plugins/username-changer/#description.

[6] An Chen, JiHo Lee, Basanta Chaulagain, Yonghwi Kwon, and Kyu Hyung Lee. Synthdb: Synthesizing database via program analysis for security testing of web applications. *NDSS*, 2023.

[7] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *USENIX Security*, 2012.

[8] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA*, 2010.

[9] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *WCRE*, 2013.

[10] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *CODASPY*, 2014.

[11] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. Black widow: Blackbox data-driven web scanning. In *S&P*, 2021.

[12] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security*, 2010.

[13] Martin Fowler. Codesmell. https://martinfowler.com/bliki/CodeSmell.html.

[14] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Backrest: A model-based feedback-driven greybox fuzzer for web applications. *arXiv preprint arXiv:2108.08455*, 2021.

[15] Google. https://security.googleblog.com/2023/02/vulnerability-reward-program-2022-year.html.

[16] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "not my bug!" and other reasons for software bug report reassignments. In *CSCW*, 2011.

[17] HackerOne. https://www.hackerone.com/reports/6th-annual-hacker-powered-security-report.

[18] Behnaz Hassanshahi, Hyunjun Lee, and Paddy Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *SANER*, 2022.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, 2004.

[20] A03:2021 Injection. https://owasp.org/Top10/A03_2021-Injection/.

[21] Joomla. https://www.joomla.org.

[22] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 2010.

[23] Faezeh Kalantari, Mehrnoosh Zaeifi, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and Adam Doupé. Context-auditor: Context-sensitive content injection mitigation. In *RAID*, 2022.

[24] Xiaowei Li, Wei Yan, and Yuan Xue. Sentinel: securing database from logic flaws in web applications. In *CODASPY*, 2012.

[25] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE*, 2008.

[26] Meta. https://about.fb.com/news/2022/12/metas-bug-bounty-program-2022/.

[27] MyBB. https://mybb.com.

[28] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'). https://cwe.mitre.org/data/definitions/79.html.

[29] OpenCart. https://www.opencart.com.

[30] Oracle. https://dev.mysql.com/doc/refman/8.0/en/information-schema.html.

[31] OWASP. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[32] Muhammad Parvez, Pavol Zavarsky, and Nidal Khoury. Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. In *ICITST*, 2015.

[33] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID*, 2015.

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *S&P*, 2018.

[35] PHPGurukul. https://phpgurukul.com/hospital-management-system-in-php/.

[36] PHPGurukul. https://phpgurukul.com/sdm_downloads/login-system/.

[37] PHPGurukul. https://phpgurukul.com/doctor-appointment-management-system-using-php-and-mysql/.

[38] PHPGurukul. https://phpgurukul.com/hostel-management-system/.

[39] Piwigo. https://piwigo.org.

[40] Prestashop. https://prestashop.com.

[41] The Menlo Report. https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf.

[42] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *CCS*, 2011.

[43] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[44] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, 2011.

[45] SCARF. https://scarf.sourceforge.net.

[46] CMS Made Simple. http://www.cmsmadesimple.org.

[47] Antoní n Steinhauser and Petr Tůma. Database traffic interception for graybox detection of stored and context-sensitive XSS. *Digital Threats: Research and Practice*, 2020.

[48] OWASP Top Ten. https://owasp.org/www-project-top-ten/.

[49] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *S&P*, 2022.

Table 4: Web applications used in the evaluation

Application	Date	Version	GitHub Stars	Lines of Code	Prior Research
CMSMS	2022	2.2.16	∼	144944	
Doctor Apt.	2023	2023/1/11	∼	65603	[49]
Hospital	2022	2022/11/8	∼	67667	[49]
Hostel	2021	2021/9/30	∼	9377	
Joomla	2023	4.2.8	4.5k	747197	[11, 33, 47, 50]
MyBB	2023	1.8.33	932	153055	[33]
OpenCart	2023	4.0.1.1	6.8k	186101	
Piwigo	2023	13.6.0	2.6k	280906	[33]
PrestaShop	2022	1.7.8.8	7.3k	1175530	[11, 47]
SCARF	2007	2007/2/27	∼	1318	[7, 11, 24]
User Login	2021	V3	∼	7036	[49]
WordPress	2023	6.1.1	17.6k	651599	[9–11, 33, 49, 50]

[50] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. webfuzz: Grey-box fuzzing for web applications. In *ESORICS*, 2021.

[51] WordPress VIP. https://docs.wpvip.com/technical-references/security/validating-sanitizing-and-escaping/.

[52] WordPress. https://developer.wordpress.org/apis/security/escaping/.

[53] WordPress. https://wordpress.com.

[54] OWASP ZAP. https://www.zaproxy.org.

A Appendix

A.1 Spider-Scents configuration

We have implemented a variety of tunable parameters for configuring Spider-Scents' choice of heuristics while scanning. Some notable parameters are:

- Avoid sensitive rows or not
- Insert rows into empty tables or not
- Configure the traversal through the database (order by table, row, column, random, reverse)
- Breakage threshold and detection type (based on status codes, response length, link content)
- Enforce independence across boundaries (across table, row, column)

We evaluate our approach avoiding sensitive rows, inserting into empty tables, traversing the database by tables and then columns, with breakage sensitive to status codes and allowing up to 50% of link content to be missing, and independence enforced across bounds.

Table 5: The number of unique database columns affected by each scanner. For each column in the table, we present: database columns only covered by Spider-Scents ($A \setminus B$), columns covered by both scanners ($A \cap B$), and columns covered by the other scanner ($B \setminus A$). The very last column presents the maximum number of columns that allow arbitrary text values.

Crawler	Arachni			Black Widow			ZAP			**MAX**
	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	$A \setminus B$	$A \cap B$	$B \setminus A$	
CMSMS	85	13	1	82	16	3	85	13	2	111
Doctor Apt.	6	10	0	14	2	0	12	4	0	16
Hospital	14	28	2	20	22	2	21	21	2	44
Hostel	15	19	0	15	19	0	7	27	0	36
Joomla	283	12	1	281	14	1	287	8	0	325
MyBB	194	15	5	133	76	15	184	25	7	264
OpenCart	282	3	0	272	13	1	278	7	0	326
Piwigo	37	21	1	41	17	2	32	26	2	63
PrestaShop	306	55	3	313	48	4	329	32	2	410
SCARF	10	5	0	2	13	0	8	7	0	15
User Login	2	4	1	2	4	1	6	0	0	7
WordPress	35	9	2	22	22	5	16	28	7	53

Table 6: Known stored XSS vulnerabilities from CVEs and other publications.

Application	Source	Description	We Find
CMSMS	CVE-2023-36970	File upload stored XSS	✗
Hospital	https://github.com/Ko-kn3t/CVE-2020-25271	username	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3	weight	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3	temperature	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3	medicalpres	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3	BloodPressure	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms3	BloodSugar	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2	PatientName	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2	PatientEmail	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2	PatientGender	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2	PatientAdd	✓
Hospital	https://sisl.lab.uic.edu/projects/chess/cross-site-scripting-in-hms2	PatientMedhis	✓
Hospital	https://www.exploit-db.com/exploits/47841	doctorspecilization	✓
Hostel	CVE-2020-25270	guardianName	✓
Hostel	CVE-2020-25270	guardianRelation	✓
Hostel	CVE-2020-25270	corresAddress	✓
Hostel	CVE-2020-25270	corresCIty	✓
OpenCart	https://github.com/nipunsomani/Opencart-3.x.x-Authenticated-Stored-XSS/blob/master/README.md	Category description	✗
SCARF	[11]	Add session	✓
SCARF	[11]	Comment	✓
SCARF	[11]	Conference name	✓
SCARF	[11]	Edit paper	✓
SCARF	[11]	Edit session	✓
SCARF	[11]	Delete comment	✓
SCARF	[11]	General options	✓
SCARF	[11]	User options	✓
User Login	CVE-2022-43097, CVE-2020-23051, CVE-2020-24723	fname	✓
User Login	CVE-2022-43097, CVE-2020-23051, CVE-2020-24723	lname	✓
WordPress	https://research.securitum.com/xss-in-wordpress-via-open-embed-auto-discovery	Embed in post content	✗